Specification

For Over-The-Air

Data Sync Protocol

V1.0

ABSTRACT

This document provides the functional and behaviour requirements of the Over-The-Air Synchronization Protocol. It provides the protocol layout and lists observable protocol scenarios.

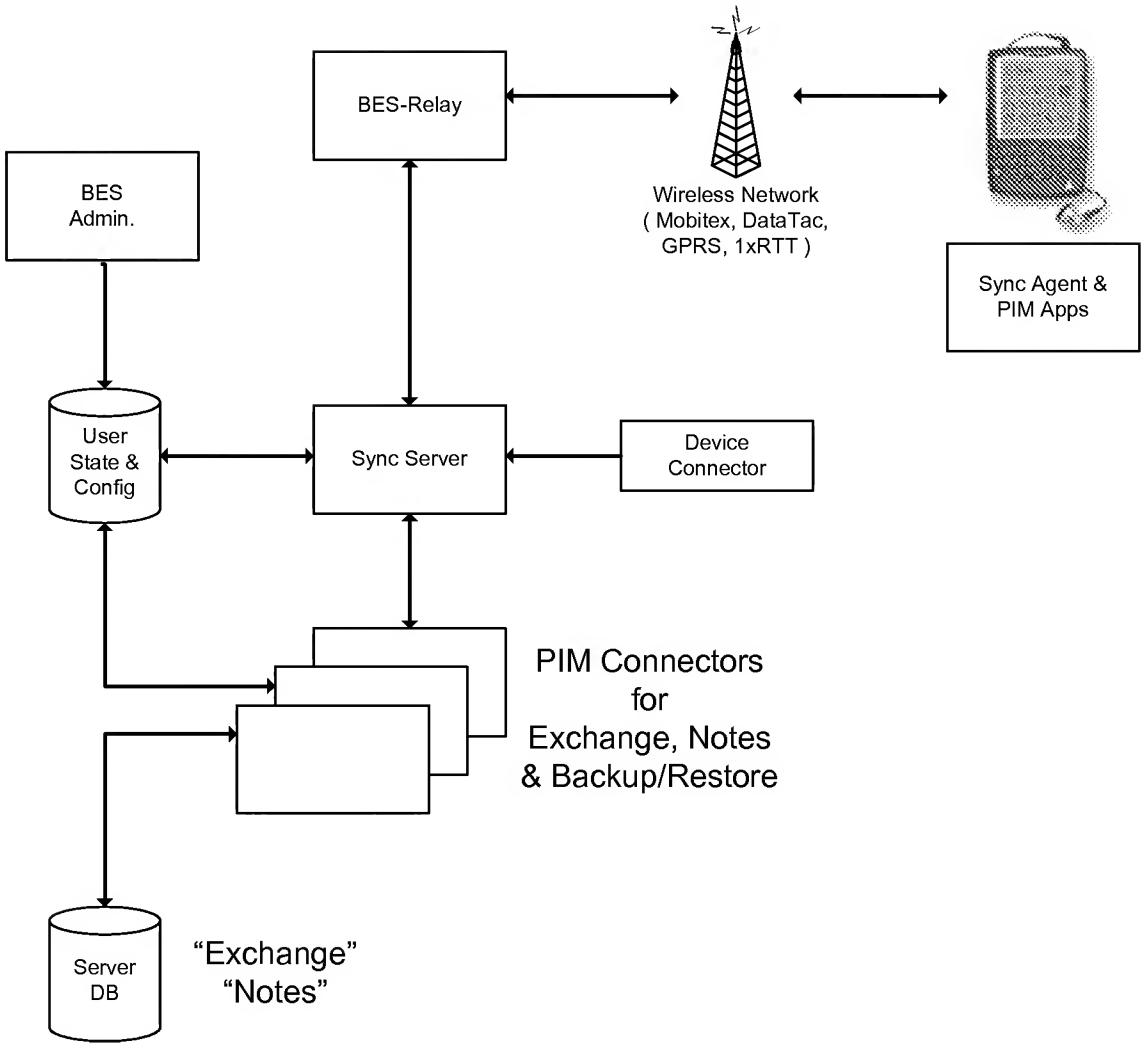
Table Of Content

1 Introduction

This document is to address the following issues related to the OTA Data Sync Protocol.

Overview of the basic components in OTA Data Sync System
Protocol Basic Concepts
Protocol State
Protocol Datagram Format
Capturing the Protocol Behavior

2 Overview of the basic components in the Sync system



Sync Server

The Sync Server provides an interface between installable connectors and the device's Sync Agent. The Sync Server accepts XML sync requests from the connectors, filters any data and converts the filtered data to tag-length format. This tag-length format along with associated header information is sent over the air to the Sync Agent. The BES Admin is periodically polled by the Sync Server for user configuration

information. If there are any configuration changes, the Sync Server then sends the new configuration to the Sync Agent and associated connectors. The Sync Server uses 'Batch' mode for storing sync requests before sending the commands to the Sync Agent.

PIM Connectors

PIM Connectors interface between a server, such as Exchange or Notes, to the Sync Server. When record changes occur within the server's DB, the connector detects these record changes, formulates an XML sync request and sends this request to the Sync Server. PIM connectors must provide an XML document that contains database schema/mapping information of a particular database on the device. The Sync Server processes this document to transform any data from the PIM connector to the device. Although the Backup/Restore connector does not provide any sync capability, the Sync Server will pass commands directly to the connector.

Device Connector

The Device Connector is a special connector that contains specific information about the device. After the XML sync request as been processed, the Sync Server passes this XML request to the Device Connector where the XML request is converted to Tag-Length format.
Commands received from the device are processed in the opposite direction, that is, the Tag-Length is converted to an XML sync request and further processed by the Sync Server.

Sync Agent

The Sync Agent, which interfaces to the PIM applications on the device, receives record change notifications from PIM applications, converts to Tag-Length format and sends this sync requests to the Sync Server. The Sync Agent must also process Tag-Length commands from the Sync Server and update any changes to the PIM applications. The Sync Agent uses 'batch' mode for storing sync requests before sending them OTA.

BES Admin

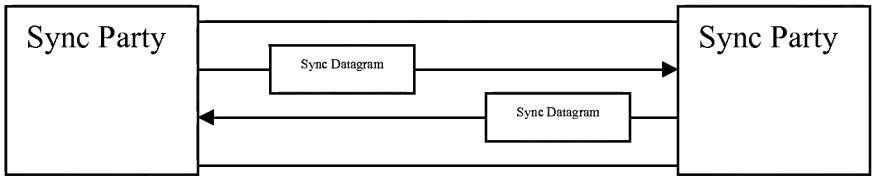
BES Admin is responsible for setting up user configuration information for the Sync Server. It provides UI configuration for setting field mapping, enabling/disabling of sync databases and various related sync

configurations. Since BES Admin provides no notification to the Sync Server, the Sync Server must poll for any user changes.

3 Protocol Basic Concepts

3.1 Sync Session

The Sync Protocol is a ***Session Oriented Protocol***. A ***Sync Session consists*** of flow of **Sync Datagrams** back and forth between **Sync Parties**. Sync Parties are the protocol implementers.



A ***Sync Datagram*** is a container that holds sync-able database changes. A sync-able database change list is a set of sync commands that reflect the changes on a certain sync-able database.

A Sync Party that starts a sync session is called a ***Session Initiator***. A Sync Party that receives sync datagrams from the Session Initiator is called the ***Session Recipient***.

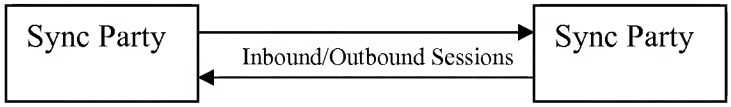
To determine whether Sync Parties are synchronized with each other, every Session Initiator must capture and include ***Session State Information*** with every sync session it initiates.

The Session State Information includes the **Change List *ID*** and ***Expected Change List ID numbers***. The Change List ID is a unique number that identifies a Change List between two sync parties. The Expected Change List ID is that is expected by the Session Initiator at the point it initiated a sync session. These numbers are used as **Sync Anchors between sync parties**.

A sync session starts when a sync party sends the first sync datagram of a particular session. A sync session does not require a connection

establishing process between sync parties involving special sync datagrams. A sync session is considered complete when the Session Initiator receives a response from the Session Recipient or it times out waiting for a response from the other sync party.
Every Session is associated with an ID called Session ID. This ID is fixed during the lifetime of a session. A started sync session waiting for a response is called an **Active Sync Session.**

If the Session is complete, then the Session ID of that particular session is considered invalid. At any one time, there is only one **Inbound** and/or one **Outbound** sync session per sync party.



3.2 Change List

All databases involved in the sync process must be associated with a change list at any point of time as long as it is part of sync process.

A Change List has broad definition in terms of what it could include.

- A Change list might contain a list of changes that happened on a database. For example, when a new record gets added to a database, the change list will hold the record information for that record and the database information. If a record is updated, the list contains the record ID of the updated record, the altered fields and database information. A Change List must be known before it gets associated with a session.
- Change List might contain initialization or request to fetch configuration.

There are two types of Changes Lists:

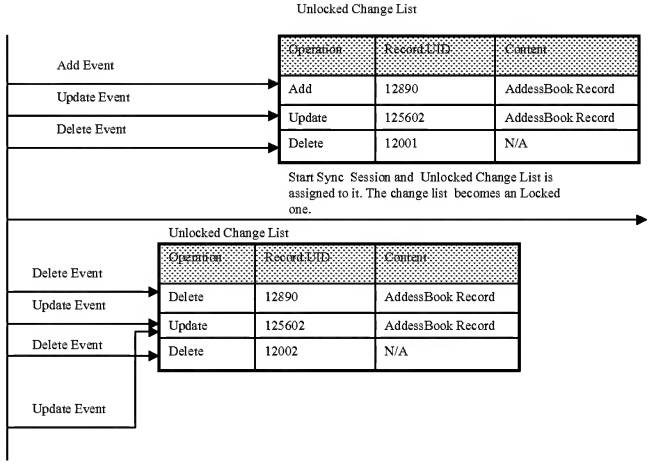
- ***Unlocked Change List***
- ***Locked Change List***

Unlocked List is a change list that is not yet associated with a sync session. This type of Change List can still accept changes from its associated database.

Locked Change List is a change list that is associated with a sync session and is currently being processed and/or being delivered to a sync party. Every locked change list MUST be given an ID called the Change List ID. This ID is fixed and SHOULD not change as long as there are elements in this list. The Change List ID is based on an incremental ID and should be equal to the ID of the last successful session sent from this initiator plus 1.

I.e., if the last Change List on device that was acknowledged by server is 6, then the next change list will be 7.

An unlocked change list becomes a locked one whenever a sync party decides to process it and send it to another sync party.


Unlocked Change List
Add Event
Update Event
Delete Event
Operation
Record UID
Content
Add
12890
AddessBook Record
Update
125602
AddessBook Record
Delete
12001
N/A
Start Sync Session and Unlocked Change List is assigned to it. The change list becomes an Locked one.
Unlocked Change List
Delete Event
Update Event
Delete Event
Update Event
Operation
Record UID
Content
Delete
12890
AddessBook Record
Update
125602
AddessBook Record
Delete
12002
N/A

3.2.1 Wireless bandwidth utilization

Due to limited bandwidth of wireless network and resource limitations on a mobile device, the database's change events should be optimized before sent over the air. To optimize the Change Lists, a few considerations regarding how these events get carried over the sync datagrams are listed below:

- All change events related to databases' record changes MUST report only the changed fields of a record.
- All string values should not be null terminated values.
- All record fields MUST be represented In Tag Length Encoding.
- All record fields MUST hold unique tags.
- No history of changes kept for a record. All the operations MUST be optimized and committed in place for a record. This implies that in any change list, there will be only one operation associated with a certain record.
- Batch database's events if possible.
- Response datagrams SHOULD NOT carry Sync State Information as it is not required;

Previous Operation	Current Operation	Action
Add	Update	Keep the add operation and apply the updates on the record
Add	Delete	Cancel the add operation on the record
Update	Delete	Delete the record
Update	Update	Combine the update operations

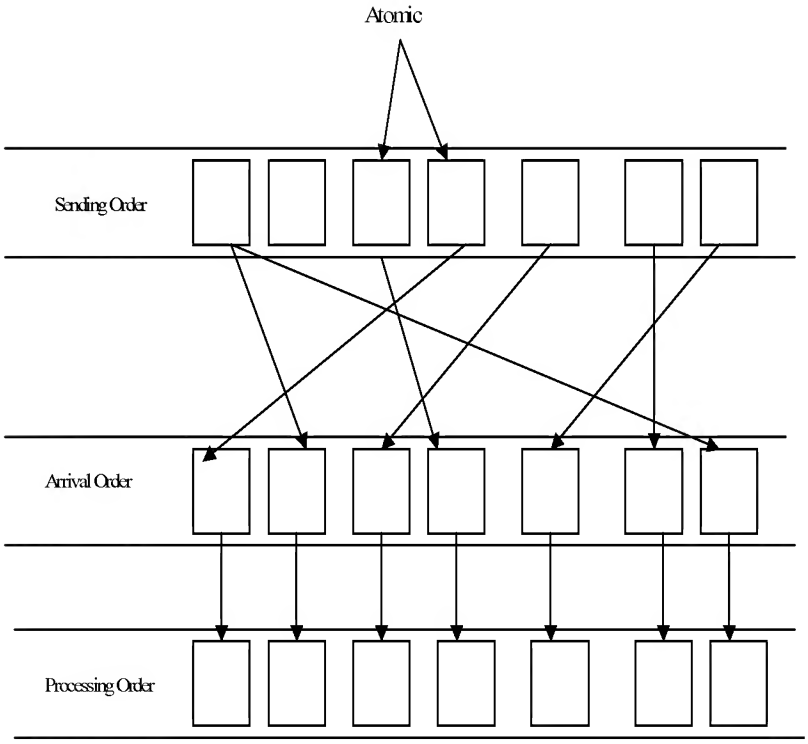
An Example of how database changes are optimized.

For example, adding and then deleting the same record could be optimized to no operation on that record. Updating a record twice will be combined to one update operation. Adding and then updating a record

operations could be optimized by merging the changes in one change that holds the add operation.

3.2.2 Protocol optimization

A Locked Change List could be split over many sync datagrams if the change list could not be carried over a single sync datagram. To improve the overall protocol efficiency, sync datagrams MUST be **Atomic**. Atomic implies that the sync parties must make the sync datagrams within a sync session independent of each other; thereby eliminating the need to queue or order datagrams as they come to a Sync Part. However, one record may be spread cross multiple datagrams. In this case, the receiver must collect fragmented record before processing the record.


Atomic
Sending Order
Arrival Order
Processing Order

Here are a few definitions that will help in defining atomic sync datagrams.

Dependent Command	A Sync Command is defined as a Dependent Command if and only if its execution depends on another Sync Command.
Independent Command	A Sync Command is defined as an Independent Command if and only if its execution does not depend on another Sync Command.
Forward Dependency Relationship of a Command	A Sync Command is defined to have a Forward dependency relationship if it depends on another Sync Command that appears later in a Sync Commands Section of a sync datagram.
Backward Dependency Relationship of a Command	A Sync Command is defined to have a backward dependency relationship if and only if The Sync Command depends on another Sync Command that appears before it in a Sync Commands Section.
Local Forward/Backward dependency relationship of a Command	A Sync Command has Local Forward/Backward dependency relationship if and only if its execution depends on another Sync Command that appeared after/before it in the same Sync Commands Section in which both Sync Commands appear.
External Forward/Backward dependency relationship of a Command	A Sync Command has External Local Forward/Backward dependency relationship if and only if its execution depends on another Sync Command that appeared before/after it in a previous Sync Commands Section or on another Sync Command that might appear later in the in a Sync Commands Section within the same session.

A sync datagram is defined as atomic sync datagram if and only if all its Dependent Sync Commands have ***Local Forward Dependency Relationship.***

4 Protocol State

The protocol state is considerably small:

- Last Change List Id Issued
- The Expected Change List ID that would be received from another sync party.

These two variables represent the sync state information.
These variables are positive integers in Big Endian format.

5 Protocol Datagram Format

Every sync datagram MUST start with a **Version** byte indicating the datagram format and the protocol behavior. All sync datagrams MUST have **Header and Sync Commands Section. For optimization purposes, the response datagrams might include the commands section if the status implies that session and its commands all were successful.**

Version	Header Section	Sync Commands Section

The Header Section starts immediately after the Version byte and ends with End Header Indicator. End Header Indicator is a byte in size and holds 0x00 value. Note the use of the < … > notation to indicate the use of Tag Length Data Encoding format. Tag Length Data Encoding Field, TLE field for short, consists of

Field TAG.
Field Length in a compressed format.
Field Value.

All the header fields are TLE fields with a specified data type.

Version	Header fields	0x00	Sync Commands section

5.1 Header Section

The following table shows the Header fields; where

- ***Field Name*** column indicates name of the field,
- ***Tag*** column indicates value TAG Field used in TLE Field,
- ***Type*** column *indicates the data type of a field,*
- ***Required*** column indicates whether a field is required or it is an optional one. {Yes} to indicate that it is mandatory in certain cases (see comments to see when it becomes a mandatory one),
- ***Default Value*** column indicates the default value of a field if it is not provided in the header.

Field Name	Tag	Type	Required	Default value	Comments
DeviceSession ID	0x01	<Integer>	{Yes}		Every Sync Session has an Id. The Session ID is a number that uniquely identifies a sync session. This field will always appear if the device initiates the session. It will not appear if the server initiates the session. 0 is a reserved value for the session id. The value of this field is always a positive one.

ServerSession ID	0x2	<Integer>	{Yes}		This field is for the server-initiated session. 0 is reserved value for the session id. The reason for having two header fields for the session ID is to eliminate the negative sign as direction indicator; which has an overhead in terms of the header size.
Current Change List ID	0x03	<Integer>	{Yes}		This field represents the current change List number that is initiated. It is not required to be sent with response datagrams for a session since Session Id is good enough to identify a session.
Expected Change List ID	0x04	<Integer>	{Yes}	0	This field represents the Change List ID that the initiator expects when the receiver creates its next session. This field is used to determine conflict resolution and obsolete changes.
Current Sequence	0x05	<Integer>	No	0	Since there is a possibility that a sync session spans over multiple sync datagrams, the *Current Sequence* indicates the current index of the sync datagram. This field will not be included when the value of the sequence is 0.

Last Sequence	0x06	<Integer>	No	0	This field is to indicate the index of the last sync datagram that in a session. NTB: the index starts at 0. The pervious field and this field help determine that all datagrams in a session have been received or not.
Verification	0x07	<Boolean>	No	N/A	This field used with the Verification step before starting a slow sync process. In the verification step this field MUST exist. The value for this header is either FALSE if the header works as a verification request or TRUE if the header was a response to a Verification Request. NO DEFAULT VALUE IS ASSUMED if this field does not exist in the header section.

The Sync Commands Section starts immediately after the ***End Header Indicator***. There is no End Indicator for the Sync Commands Section of a sync datagram since the end of the Sync Commands section is the end of the sync datagram provided by the sync transportation layer.

Note that all response datagrams

- MUST contain either Device Session Id or Server Session Id but not both.
- May contain Verification header field if the datagram is meant for verification step before a slow sync starts.

5.2 Sync Commands Section

The Sync Commands Section in the sync datagram consists of a list of sync commands. The order and execution of sync commands is **IMPORTANT** since some of Sync Commands have dependency with other sync commands.

A sync command may contain a list of parameters. Any parameter that exists in the Commands Parameters List of a sync command **MUST** be in Tag Length Value Encoding format.

Sync Command Tag	Command's Parameters List

The following table has notations used in the next sections to help clarify the Sync Commands and Sync Commands parameters used.

Optional (argument)	To represent an optional argument.
Array (argument)	To represent an array of an argument.
Either (argument1,argument2,…);	To represent a single choice between some arguments.
Pair (argument1, argument2, …)	To represent a pair or more of arguments together.

For example, ***Array (Pair (CmdId, Result Code)***) represents a sequence of parameters of a Pair of CmdId and Result Code.

5.2.1 Sync Commands' Parameters

Parameter Name	Parameter Tag	Parameter	Comments
Record.UID	Ox01	<Integer>	Every record has a unique number that is unique per database.
Record.Fields	0x02	<Byte []>	The record fields using Tag-Length encoding. The order of theses fields is not important.
Record.KeysHash	0x03	<Integer>	A 4-byte value representing the 'key' fields of a record. The order of the fields is important when calculating the hash value. The order would be based on the database schema.
Record.FieldsHash	0x04	<Integer>	A 4 byte value representing all mapped fields. The order of calculating the hash value is important The order would be based on the database schema.
DataSource.ID	0x10	<Integer>	A number that is used instead of the Data Source Name. It is mainly used for optimization purposes.

Database.Name	0x20	<String>	A String that identifies the database.
Database.ID	0x21	<Integer>	A number that identifies the database for optimization purposes.
Session.DatagramSequence	0x30	<Integer>	To represent the sequence number of a Sync packet
Sync.Configuration	0x40	<Byte[]>	Please see the Configuration Settings Appendix
Cmd.ID	0x50	<Integer>	The value is defined as follows: BIT 0-15: Index of a command. BIT 16-31: Datagram sequence number.
Version	0x60	<Byte>	Version parameter
Index	0x61	<Integer>	Index parameter
LastIndex	0x62	<Integer>	Last Index parameter
ErrorCode	0x63	<Short>	To Represent an Error Code
Group.ID	0x70	<Byte>	This field represents The Group Id of a collection of records in a database.

IncludeKeyHashForNegativeUIDs	0x71	<Boolean>	Used as indicator from the server that it does not have history for records. It is only required during the slow sync process. Default value for this parameter is False.

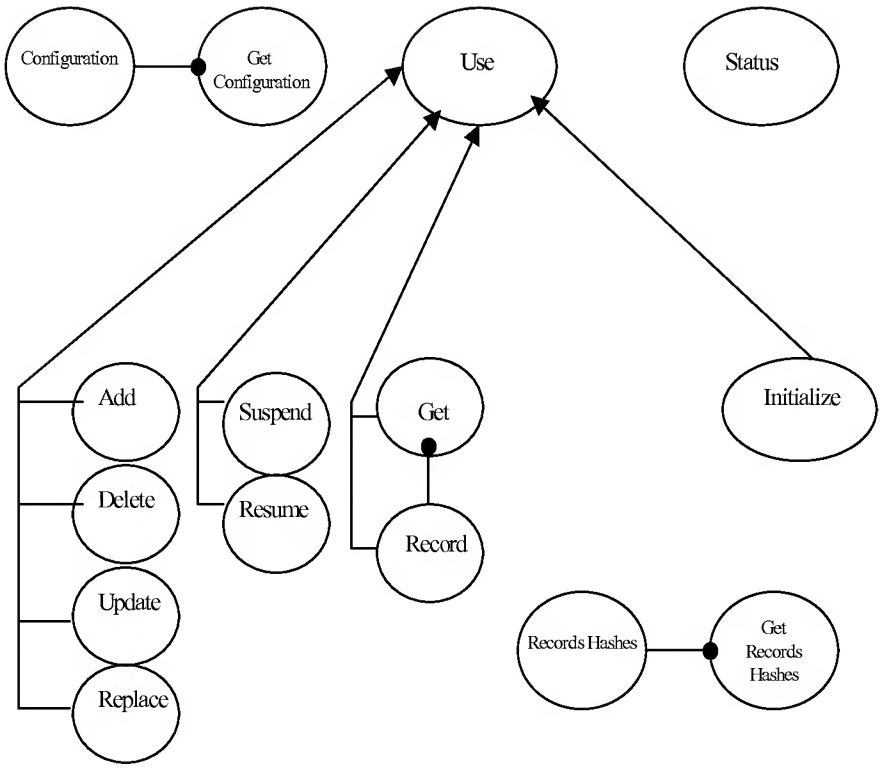
5.2.2 Sync Commands

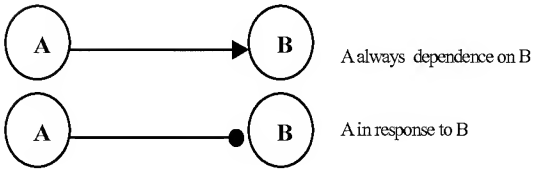
Command Name	Command Tag	Parameters	Comments
Add	0x01	Pair(Record.UID, Optional(Index), optional(LastIndex), Record.Fields)	Used for adding a record to a database specified by ***Use Command***. Index and Last Index are used for fragmented add sync commands. This command is sent from device and server.
Delete	0x02	Optional(Record.UID)	Used for deleting a record from a database specified by ***Use Command***. If the parameters list is empty, this will delete all the records within a specified database. The result of this sync operation is Always success. This command is sent from device and server.
Update	0x03	Pair(Record.UID, Optional(Index), Opional(LastIndex), Record.Fields)	Used for updating a record in a database specified by ***Use Command***. If the record does not exist, an error is returned. This command is sent from device and server. This command is sent from device and server.
Replace	0x04	Pair(Record.UID, Optional(Index), Optional(LastIndex), Record.Fields)	Used for replacing a record in a database specified by ***Use Command***. If the record does not exist, an error is returned. This command is sent from device and server.
Get	0x05	Optional (Record.UID)	Used for fetching a record, a list of records or all records from a database. ***Use Command*** MUST be used before using this command. To fetch all the records in a database, the parameter list must set to be NULL. This command is sent from device and server.

Record	0x06	Pair(Record.UID, Optional(Index), Optional(LastIndex), Record.Fields)	Used in response to a 'Get' command (see above) for sending record(s) from a database. ***Use Command*** MUST be used before using this command. If a record does exist, an error should be returned. This command is sent from device and server.
Use	0x07	Pair (DataSource.ID, Either(Database.Name, Database.ID))	Used for switching paths between databases sync commands. This command is sent from device and server. Database.Name parameter would be sent if the Database.ID is not available through the configuration. Note that either Database.Name or Database.ID iis used in this command but not in both.
Status	0x08	Optional(ErrorCode) Optional(Array(pair(Session.DatagramSequence,ErrorCode))) Optional(Array(pair(CmdID,ErrorCode)))	Used for sending session results of all sync commands within a session. This command is sent from device and server.
Get Configuration	0x09	N/A	Used for getting the configuration of the OTA sync protocol. The command is generally sent from the device.
Configuration	0x0A	Sync.Configuration	Used in response for the **GetConfiguration** command. The Sync Server formulates configuration information in tag-length format and sends this information to the device. See appendix for more information.

Initialize	0x0B	Optional (Array (group hashes) Every entry in this array would be in Tag Length Encoding where the TAG represents ID of a group. An example of an entry in this array would be TAG Length Hash Value. [0x00][[0x4]][0x34][0x98][0x78][0x67] First byte is the Group Id. The second byte is the Length of the hash of the group. And the rest of the bytes are the Hash value in Big Endian format. Note that the number of groups cannot be more than 256 groups per database. Therefore, the group ID ranges from 0 to 255.	Used for initiating a sync process for a database. This command initiated from either the device or Sync Server. If the Server sends this command, no parameters would be included. If this command sent from device the parameters would be empty if the database has no records at all. If the databases have records, then parameters will be an Array of the group hashes. For information on how the groups are created and how hashes are generated hashes, see how slow sync section.
Suspend	0x0C	N/A	Used for suspending a sync process. Use the 'Resume' command to resume operations. This command initiated from either the device or Sync Server. NOT IMPLEMENTED YET.
Resume	0x0D	N/A	Used for resuming a sync process in which a process was suspended. This command initiated from either the device or Sync Server. NOT IMPLEMENTED YET.
Get Record Hashes	0x10	Optional(IncludeKeyHashForNegativeUIDs) Array(Group.ID)	Used for requesting hashes of records during a slow sync. ***Use command*** must be used before using this command.

Record Hashes	0x11	Array (Pair (Record.UID, Optinal(Record.KeysHash), Record.FieldHash))	Used in response to 'Get Record Hashes' command. ***Use command*** must be used before using this command. Record.KeysHash might not get included if the Include KeyHash For Negative UIDs Parameter was false in the GetRecordHash command.





Sync Commands Dependency Relationship

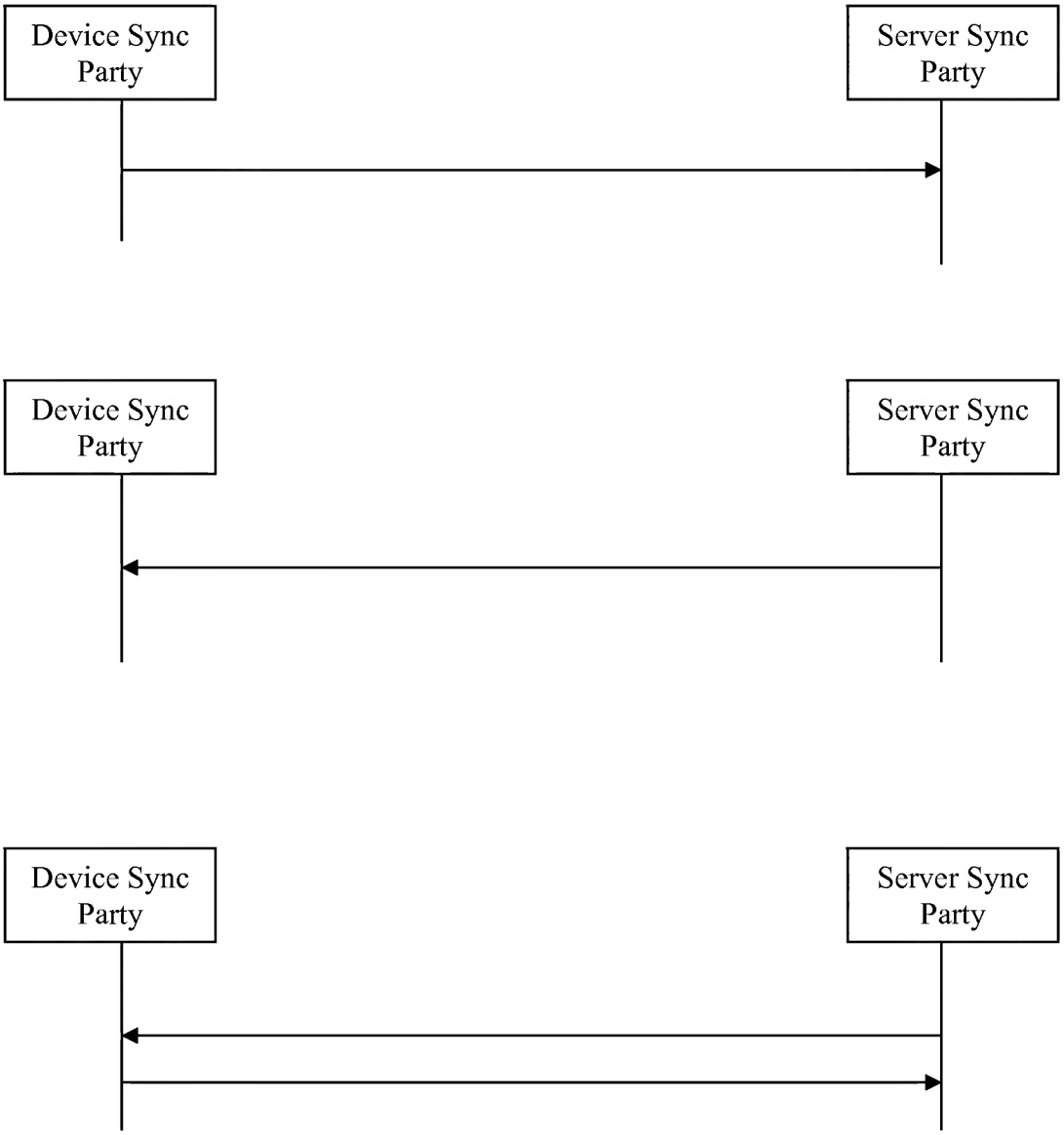
6 Capturing the Protocol Behaviour

As mentioned before, the protocol is session based. In order for a sync party to request sync changes to another party, the initiating party must start a sync session.

The sync party that starts a session is called a Session Initiator, and the sync party that receives a session is called the Session Recipient. The Session Recipient will handle the request of the Session Initiator if and only if both sync parties are in sync state when the request arrives at the Session Recipient.

To determine whether sync parties are in sync state, **Sync State Information** would have to be maintained between any two sync parties. Thus, session information would be included in every sync datagram and embedded in a sync datagram header. The exception would be the response datagrams where Sync State Information is not present since it is not required.

The Change List ID and Expected Change List ID are the header fields that carry the sync state information within a sync session. The values of these fields reflect the sync state of the sync session initiator at the point of initiating the session.



Possible Session Directions

Since both sync parties could be Session Initiators at the same time, the direction becomes an important issue when it comes to identifying which sync datagram belongs to which sync session.

There are two possible tags for the session id: server session ID and device session ID. The tag reflects the initiator. One of these two tags must be present in the header but not both.

The Change List ID and Expected Change List ID are sequential positive integer numbers. They do not indicate the direction rather they indicate a change list number of some sort that is associated with a session ID.

6.1 Notations and Definitions

To define a procedure to determine whether a Device Sync Party is In Sync State with Server Sync Party, consider the following definitions and notations.

SP = {Server Sync Part, Device Sync Party} Where **SP** represent a set of possible Sync Parties.

- **Field (T, V)** to represent a TLE field that has Tag T and Value V
- **Record (UID) = {Field (T1,V1), Field (T2,V2), Field (Ti,Vi),..., Field (Tn,Vn) }**

Where ***Record (UID)*** represents a record that has a unique ID, UID, and set of **Fields.**

- ***ChangeList(X,Y) = { Record(UID1)), Record (UID2), ….., Record (UIDn) }***
- Where ***ChangeList(X,Y)*** represents a change list of records that is defined on a database named ***X.*** and the change list is generated by ***Y,*** where ***Y E SP.***

A ***E*** B	Used for defining that an element ***A belongs to or A is part of set B.***
A ***!E*** *B*	Used for defining that an element ***A do not belong to or A is not part of set B.***
A ***I*** B	Used for defining intersection between set A and set , with the condition that all the
A ***C*** B	*Used for defining that the item A conflicts with the item B.* Record(UIDi) C Record(UIDj) if and only if • UIDi == UIDj • Record(`UIDi) *E* ChangeList(X, SPa) • Record(UIDj) *E* ChangeList(X, SPb), Spa *E* SP, Spb *E* SP and Spa != SPb • Record(UIDi) *I* Record(UIDj) != {}
A ***M*** B	The notation ***A M B*** is used for defining a merge set of two sets **A** and **B with the condition that if *A I B != {}, then if for every X E (A I B) , X E A and X E (A M B)***
A – B	The notation ***A - B*** is used for defining the delta set of two sets **A** and **B.**

6.2 Protocol Phases

There are three protocol phases in the data sync protocol. First of these is configuration which is meant to provide configuration information mainly about sources and databases involved in the sync process. The configuration information includes the sync type, sync mode and mapped fields of every database involved in the sync process.

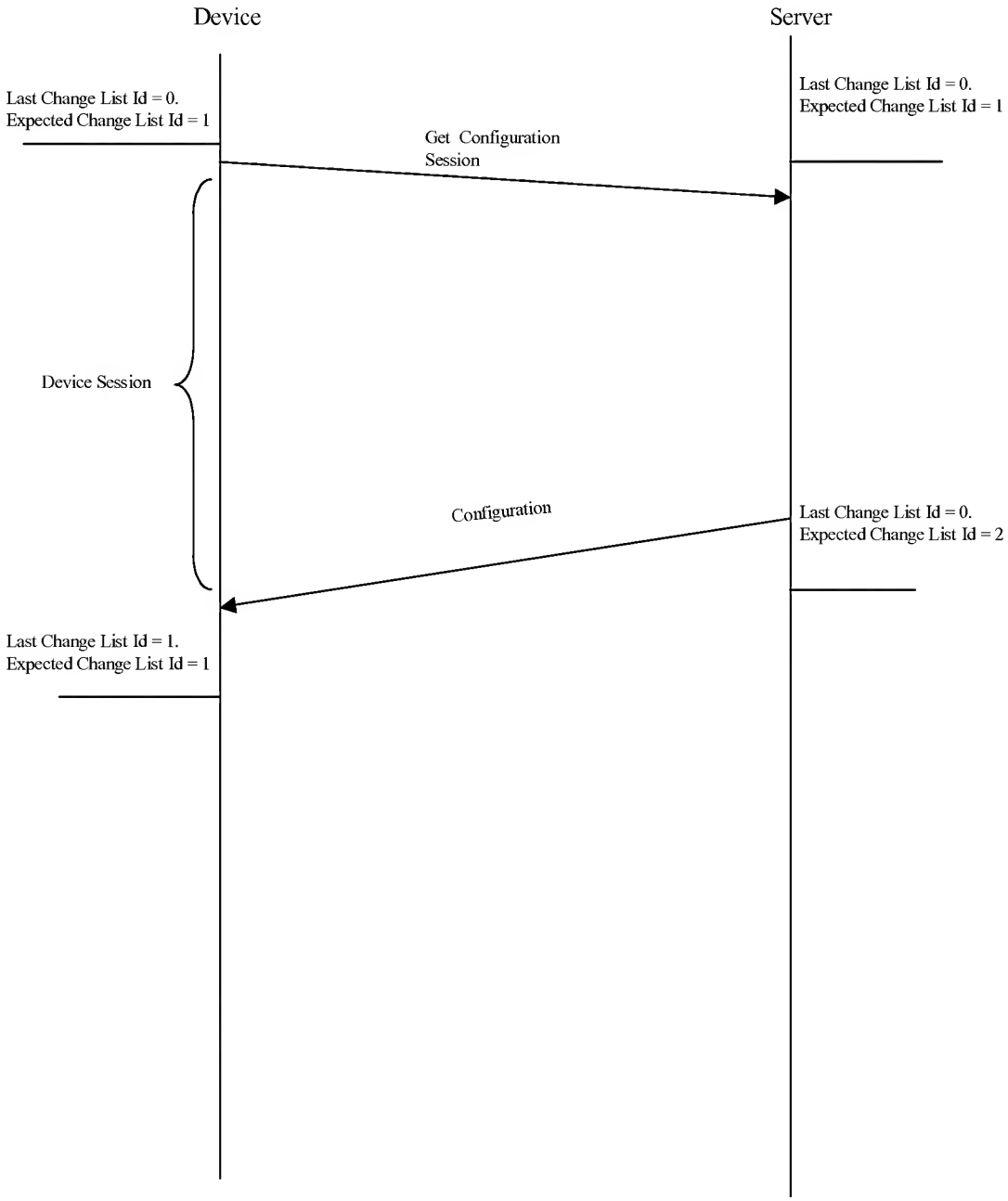
Once the configuration state is known for each database, databases must get initialized. The database initialization phase simply puts the databases in a state to be ready to accept sync operations from the device sync party or the server sync party.

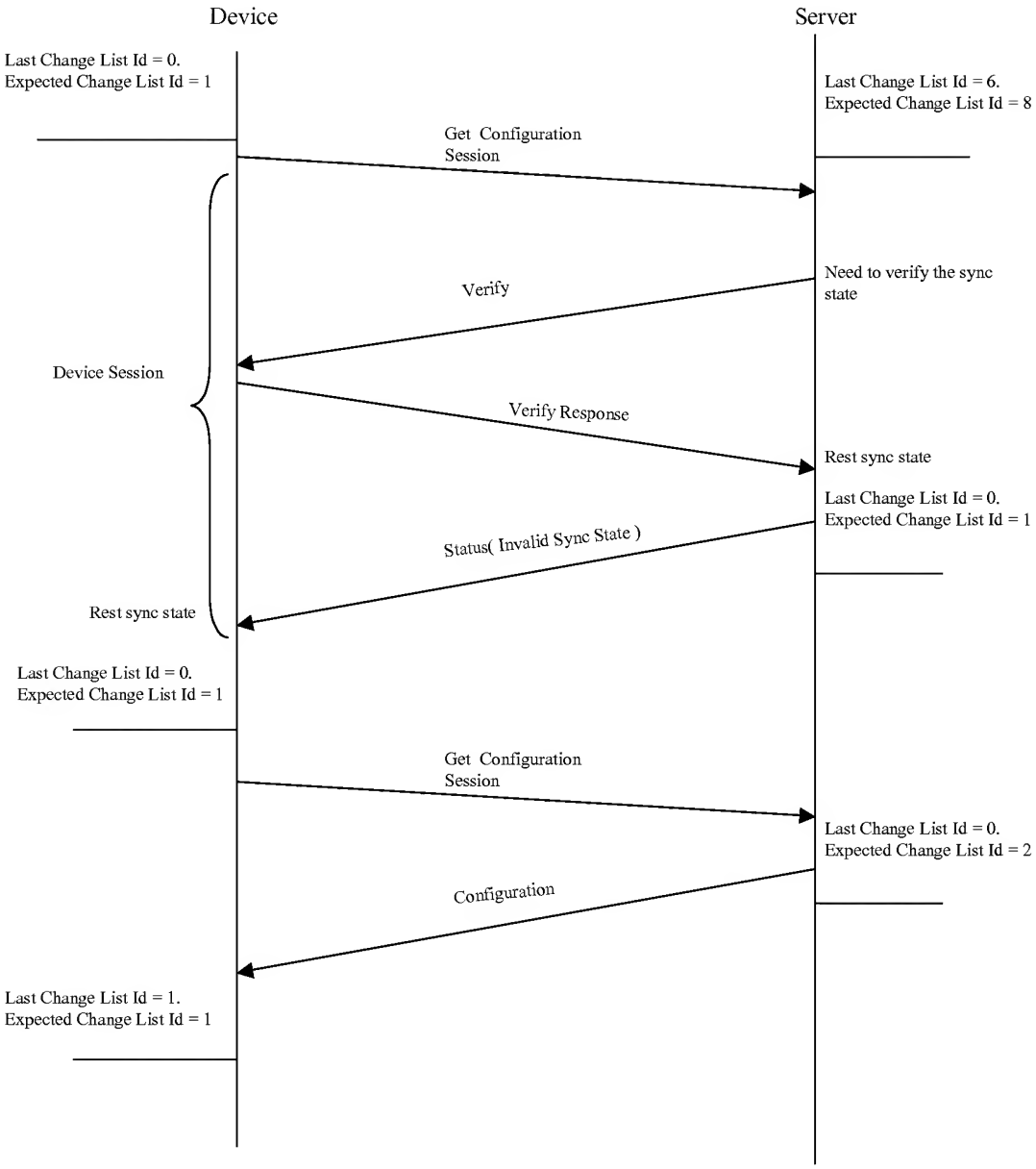
Once the databases get initialized and are ready to accept changes, the data sync phase comes in the picture. In this phase the database will be receiving data and providing notification of sync changes (such as adding, updating and/or deleting records).

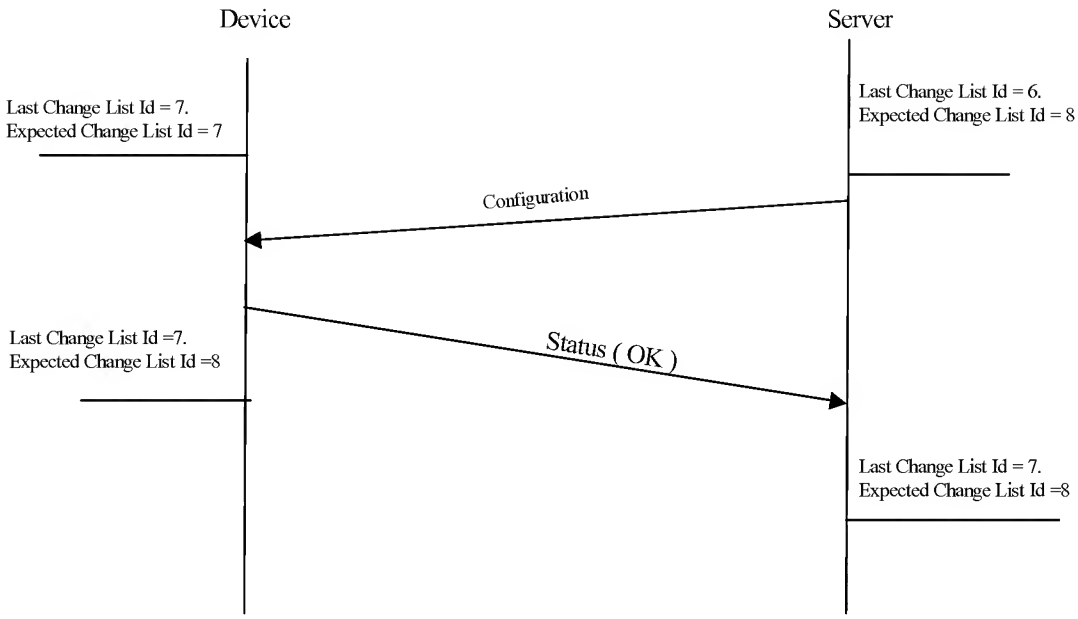
6.2.1 Configuration phase

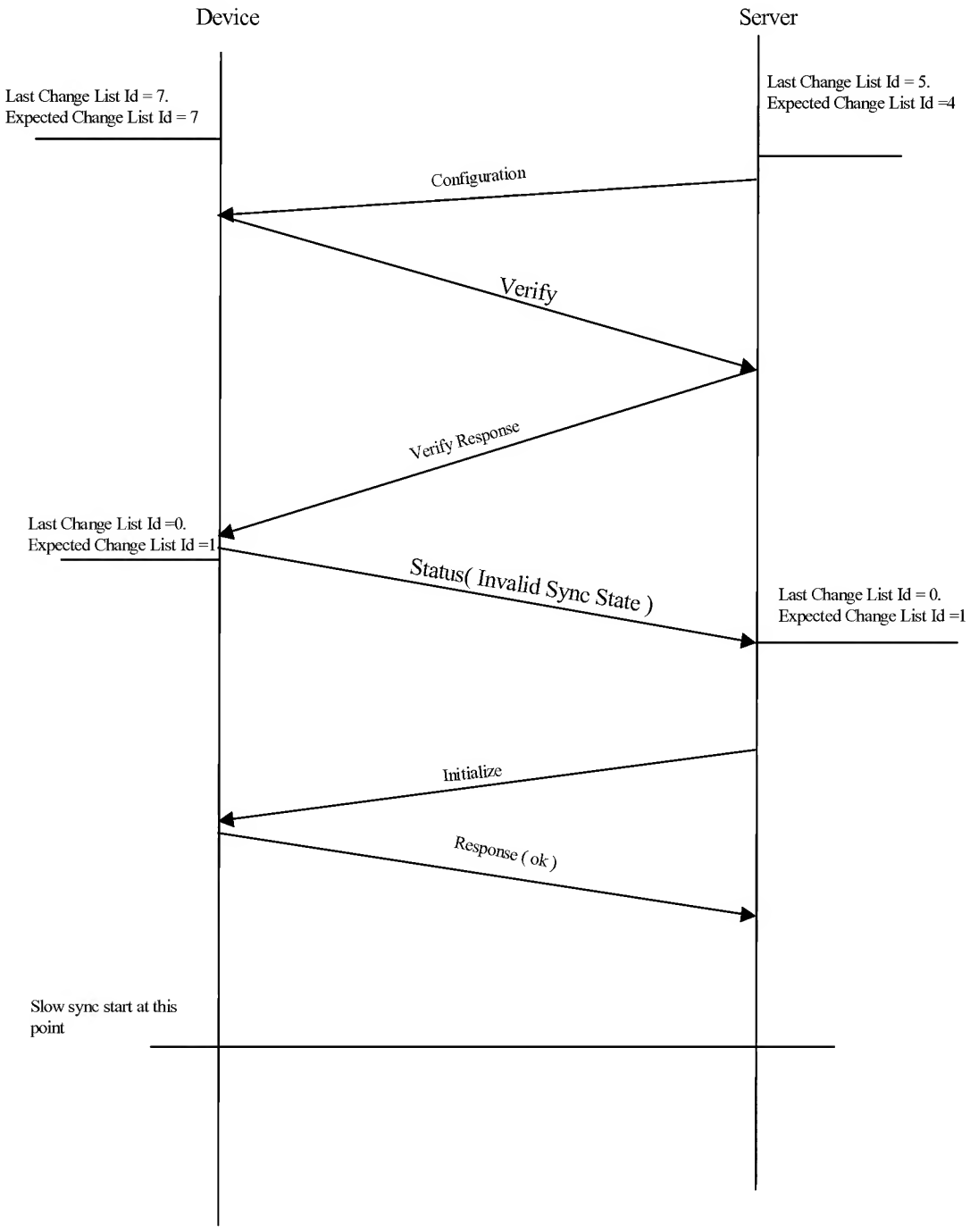
The configuration phase is always initiated from the device side whenever the device sync party detects the existence of a new Service Book with "SYNC" services. The device requests the configuration information from the sync server party, which contains a list of available data sources, databases and any other user configuration information about the sync process.

The Sync Server might update the configuration OTA by sending configuration as soon as it detects the change at the server side.


Device
Server
Last Change List Id = 0.
Expected Change List Id = 1
Get Configuration
Session
Last Change List Id = 0.
Expected Change List Id = 1
Device Session
Configuration
Last Change List Id = 0.
Expected Change List Id = 2
Last Change List Id = 1.
Expected Change List Id = 1


Device
Server
Last Change List Id = 0.
Expected Change List Id = 1
Last Change List Id = 6.
Expected Change List Id = 8
Get Configuration
Session
Need to verify the sync
state
Verify
Device Session
Verify Response
Rest sync state
Last Change List Id = 0.
Expected Change List Id = 1
Status(Invalid Sync State)
Rest sync state
Last Change List Id = 0.
Expected Change List Id = 1
Get Configuration
Session
Last Change List Id = 0.
Expected Change List Id = 2
Configuration
Last Change List Id = 1.
Expected Change List Id = 1


Device
Server
Last Change List Id = 7.
Expected Change List Id = 7
Last Change List Id = 6.
Expected Change List Id = 8
Configuration
Last Change List Id =7.
Expected Change List Id =8
Status (OK)
Last Change List Id = 7.
Expected Change List Id =8


Device
Server
Last Change List Id = 7.
Expected Change List Id = 7
Last Change List Id = 5.
Expected Change List Id =4
Configuration
Verify
Verify Response
Last Change List Id =0.
Expected Change List Id =1
Status(Invalid Sync State)
Last Change List Id = 0.
Expected Change List Id =1
Initialize
Response (ok)
Slow sync start at this point

6.2.2 Initialization phase

In this phase the device sync party starts the Initialization phase involving the slow sync process for each of the enabled databases that has a schema. The slow sync process between device and the server is done per database a time.

In order to reduce the number of records fetched from the device or sent to the device during the slow sync process, Record-grouping approach is used in which every group of records is represented by hash value. The hash value of a group represents an over all hash of a group of records.

During the slow sync process, the server will try to match the hashes of these groups sent from the device with the hashes of the groups created at the server to figure out which of the device records need not be fetched to server, which of the server records need to be sent to device and which of the records are conflicted and need to be resolved.

6.2.2.1 Grouping records

Every group is given an ID. The group ID ranges from 0 to Number Of Groups – 1. Every record has always a unique ID within a database.

To achieve better matching between server and device groups, grouping is based on this unique ID of the record; where the relation of the record UID to the Group ID is determined by the following.

- Group ID = Record UID MOD Number Of Groups.

Therefore if record UID is known, then the Group the record belongs to is known as well. The only issue with this method is that using the mode operation does not guarantee good balance in terms of the number of number of record per group.

6.2.2.2 Generating Group Hash

Generating group hash is based on all records that belong to the group. To make the calculation of the group hash an easy process, the following assumptions is considered.

- Records order does not matter within the group. Thus no record sorting is required to calculate the group hash and no matter how records arranged within the group, the group hash should be the same with every arrangement.
- The Record hash is based on the hash of the record fields. And no matter how record fields get arranged, the record hash should be the same. This made possible since all the record fields tags are unique by definition.

Thus it does not matter how record's fields appear in a record and records appear in the group, the group hash is the same. The hash function defined on the record fields is XOR. And the hash function defined in the records is as well XOR.

- **Calculating the record hash**

Since there is high chance that two records can be identical in terms of their content and the hash function applied on the records is XOR, then prefixing the record UID to the record fields hash would be a choice to reduce the chance that two records might cancel each other during applying the XOR operation them. However prefixing the record UID does not prevent the cancellation operation but just reduce it.

Note that in the hash sizes are not yet fixed.

```
// Result of this is 32 bit hash value.
int Function CalculateRecordKeyFieldsHash(Record R)
Begin
        DEFINE int hash = 0;
        For Every KeyField F in R Do
        Begin
                hash = hash XOR ((F.TAG << 16) | SH1(F.Value, 16 bits ));
        End
        Return hash;
END

// Result of this is a 32 bit hash value
int Function CalculateRecordFieldsHash (Record R)
Begin
        DEFINE int hash = 0;
        For Every Field F in R Do
        Begin
              hash = hash XOR ((F.TAG << 16) | SH1(F.Value, 16 bits ));
        End
        Return Hash;
END

Void Function AddRecordToGroup( Group G, Record R )
Begin
      G.hash = G.hash XOR SH1( (R.UID  << 32 | CalculateRecordFieldsHash( R )), 32 );
End
```

When the device sends to server an Initialize command along with an array of group hashes, the server side, upon receiving the Initialize command, will try to match the group hashes received with the hashes that it has created as well using the same method as explained before.

The Server matches groups together based on the Group Id. I.e the server will match hashes only for groups that has the same Group Ids. If hashes of two groups match, then the server considers all records that belong to that group exist on the server as well. (There is a chance, highly unlikely to happen, that two groups match, but in fact that both do not have identical records: **number of record in a group might get added when the device sends the group hash to server**).

If the two groups do not match, then they must be different from each other in terms of number of records or records' content. In that case the server will ask the device to provide keys and fields hashes for every record in the groups that do not match.

Note that if the server indicated that it did not have a history, having sent IncludeKeyHashForNegativeUIDs with the GetRecordHashses command, the KeyHashs for negative UIDs as well as the positive ones will be sent off the device. Otherwise, only the Key Hashes of records that have positive UIDs will be sent of the device.

If the server has history, then matching records would be based on UID singe.

- If the UID of a record is negative, then only UID is used for matching records.
- If the UID of a record is positive, then Key Hash is used for matching two records.

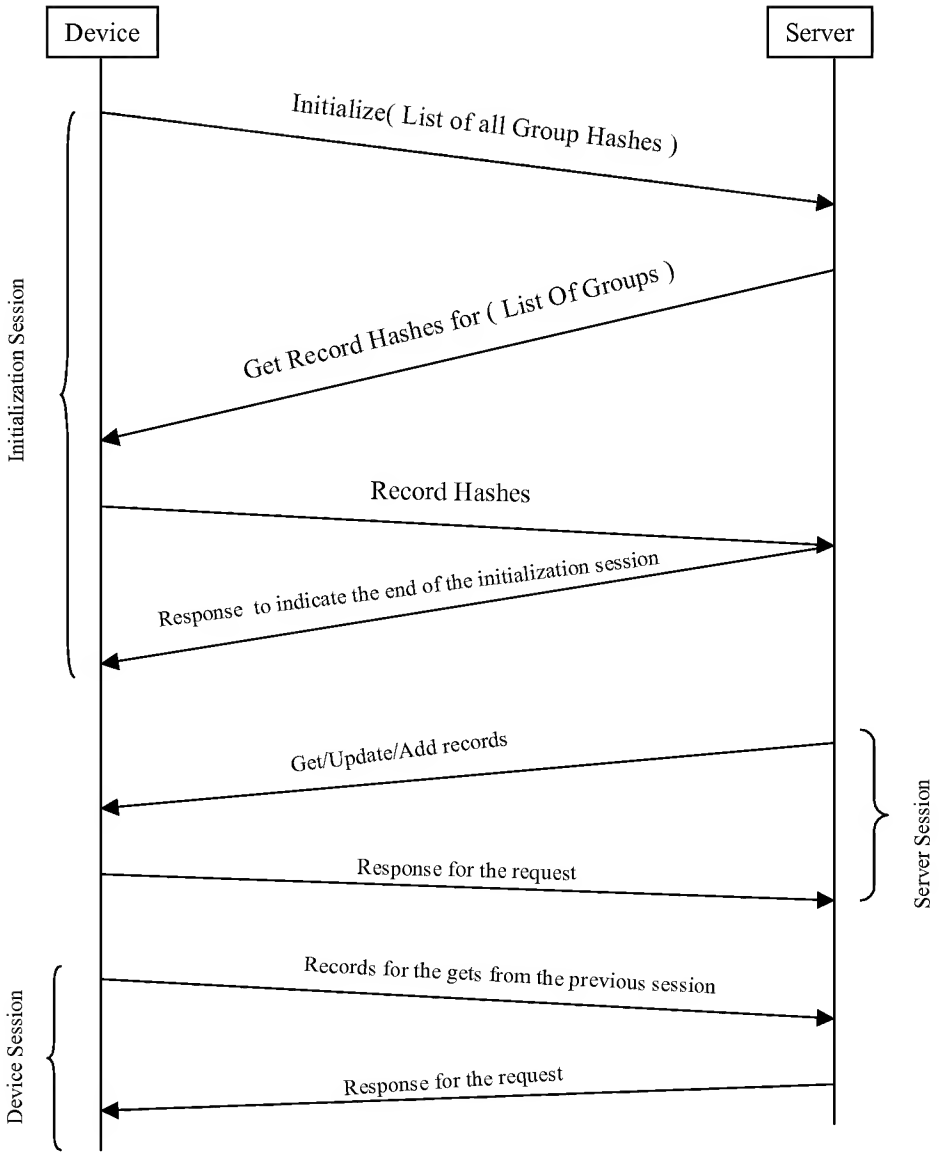
If the Server does not have history, then matching records would be always based on the Key Hashes of records.

The soul purpose of the Record Field Hash is to determine whether a record is conflicting with one another provided the two records match has the same UID and share the same key hash value in first place.

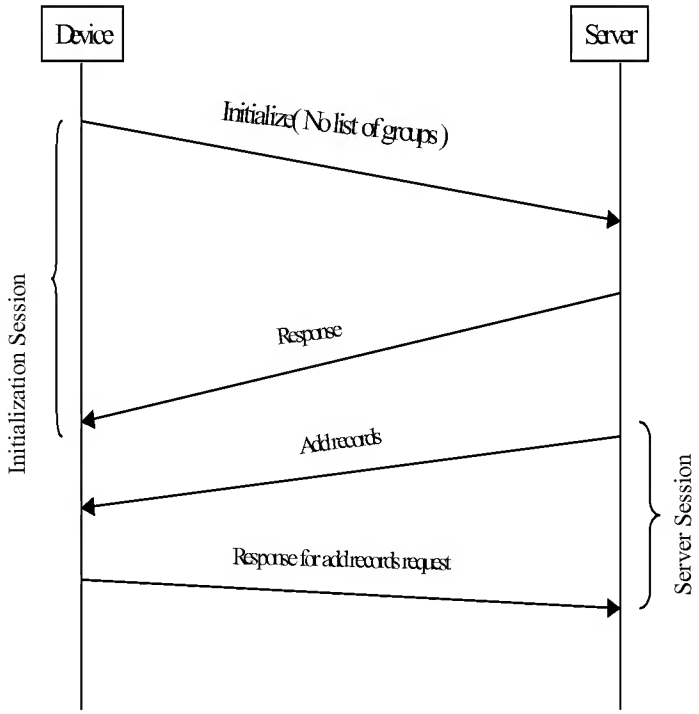
In the matching process, the server figures out which of records will be

- Added to device if they exist only on the server.
- Updated on the device if they exist in both but the server records win over the device records.
- Fetched if either exist only on the device or exist on both but the device records win over the server records.

During the matching process the server would be able to update/construct the history of records.


Device
Server
Initialize(List of all Group Hashes)
Get Record Hashes for (List Of Groups)
Record Hashes
Response to indicate the end of the initialization session
Initialization Session
Get/Update/Add records
Response for the request
Server Session
Records for the gets from the previous session
Response for the request
Device Session

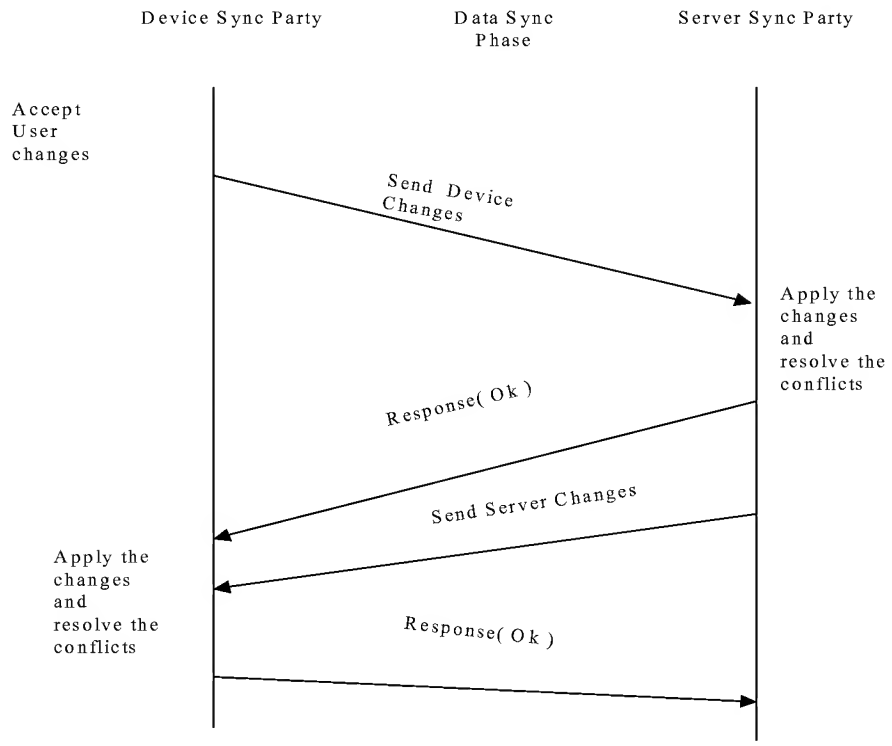
If a database device is trying to initialize does not have a record, then the initialization command would have no parameters when get sent to server. At server side, upon receiving an initialize command that has no parameters, the sync server immediately send a response to the initialization session and then, will try to send all records for that database in the following session(s) to device. At the same time the device should be able to add/update and delete records while the server adding the records.



Note: Depending on 'AllowOTASlowSync' flag, the Initialize command is deferred from being sent to the Sync Server until the device is cradled. This prevents users from initiating slow syncs over the air.

6.2.3 Data Sync Phase

After the sync-able databases are initialized, sync parties are ready to apply the sync commands on them or collect the changes and send them to the other sync party.



6.2.3.1 Sync Party State

A sync party can be in one of the following states. In any state, there may be at most one Unlocked and/or one Locked Change List.

State Number	Initiated Session	Received Session
1	No	No
2	No	Yes
3	Yes	No
4	Yes	Yes

In state 1, the sync parties are idle. Thus, both sync parties are in a waiting state.

In state 2, the sync party works as Session Recipient only. In this state, the sync party has to detect whether it was in the sync state with the Initiator Sync Party or not. Using the sync state information maintained at the recipient's side and by comparing it to the sync stat information coming along with the initiated session, the sync party could determine whether in sync state with the session initiator or not. If the Recipient Sync Party determines that it is not in a sync state with the Initiator Sync Party, the Recipient Sync Party will verify first and if necessary will respond with an invalid sync stat error.

If the Recipient Sync Party determines that it is in the sync state with the Initiator Sync Party, then it will apply the changes. The recipient also has to handle conflicting changes and respond with the status of the applied changes.

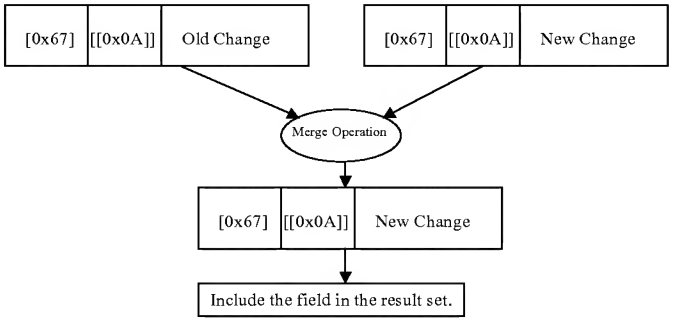
If the Recipient Sync Party is working on an active session already, then any sync datagram received with a different session ID will be ignored since only one active sync session is allowed to be initiated from a sync party at a time.

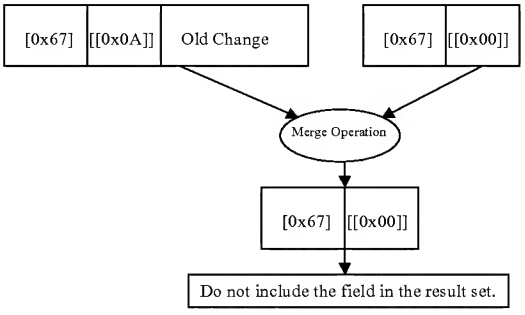
If the Recipient Sync Party is in this state where it is in the midst of processing an incoming session, it should not attempt to initiate its own session as well in order to keep the sync scenarios less complicated.

The Recipient Sync Party MUST have at most one Unlocked Change List and no Locked Change List pending. If a Recipient Sync Party has one Unlocked Change List, then a conflict may exist between the Recipient Sync Party Changes and the other Initiator Sync Party Changes.

Currently all conflicts happen only at the field level changes rather than the complete record. As a result, conflict policies must be applied on field level rather than on a record level.

Note that conflict resolution process requires applying merge operation between records. When applying a 'merge records operation', a field would be deleted from the merge result set if it has a length of zero.





To resolve the conflict between the Recipient Sync Session and the Initiator Sync Session.

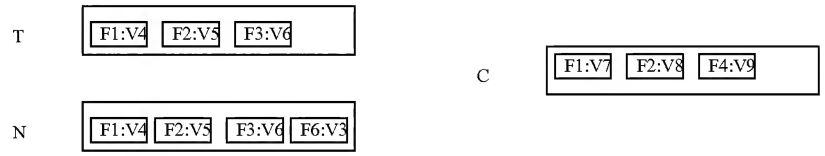
Assume:

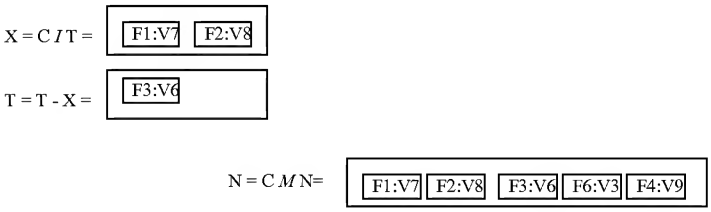
T = Record (UIDi) *E* Unlocked Change List (X, SP(a))
C = Record (UIDj) *E Locked* Change List (X, SP(b))
The changes made in SP(b) will win over the changes made in Spa
N *E* X in SP(a).

Where (SP(a), SP(b) *E* SP), (T, C and N *E X*) *and* (UIDi == UIDj).

Apply:
X = C *I* T
T = T - X
N = C *M* N

Example,


T
F1:V4
F2:V5
F3:V6
C
F1:V7
F2:V8
F4:V9
N
F1:V4
F2:V5
F3:V6
F6:V3


X = C I T =
F1:V7
F2:V8
T = T - X =
F3:V6
N = C M N=
F1:V7
F2:V8
F3:V6
F6:V3
F4:V9

Note that in this example, after the conflict resolution process, the unlocked changes get overridden by the session initiator's changes even though the recipient changes may win over the initiator changes. This happens because the recipient changes have not been sent yet.

Assume:

T = Record (UIDi) *E* Unlocked Change List (X, SP(a))

C = Record (UIDj) *E* Locked Change List (X, SP(b))

The changes made in SP(a) will win over the changes made in SP(b)

N *E* X in SP(a).

Where (SP(a), SP(b) *E* SP),(T,C and N *E* X) *and* (UIDi == UIDj).

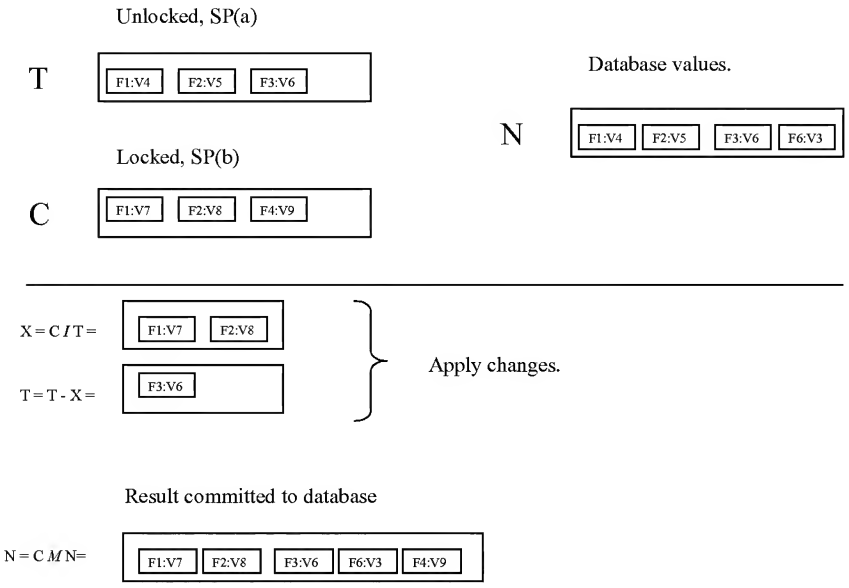
Apply:

X = C *I* T

T = T - X

N = C *M* N

Example,



In state 3, the Sync Party works as the Initiator Sync Party, sending sync state information along with the sync datagrams to the Recipient Sync Party, and waiting for a response from the Recipient Sync Party.

When a response from that session is received by the Initiator Sync Party, it will update its sync state information by incrementing the Change List value by one.

If the Initiator Sync Party times out waiting for the response for its session, it should then terminate the current session and try to open a new session for resending the changes again along with the sync state information.

In state 4, the sync party works as Initiator and Recipient Sync Party simultaneously. The sync party in this state must be able to handle conflicts between its changes and the other sync party changes. What applies in state 2 and state 3 now has to apply again with the consideration of how to resolve the conflicts on a locked change list. If the Sync Party was Server Sync Party, then

To resolve the conflicts:

Assume:

A = Record (UIDi) *E* Locked Change List (X, SP(a))

T = Record (UIDj) *E* Unlocked List (X, SP(a))

C = Record (UIDk) *E Locked* Change List (X, SP(b))

The changes made in SP(b) will win over the changes made in SP(a). N is the original record that holds the most updated version of the conflicted record, which most likely resides in a database.

Where SP(a), SP(b) *E* SP and X is the database that A, T and C belongs to and UIDi == UIDj == UIDk.

Apply:

X = C *I* A

Y = C *I* T

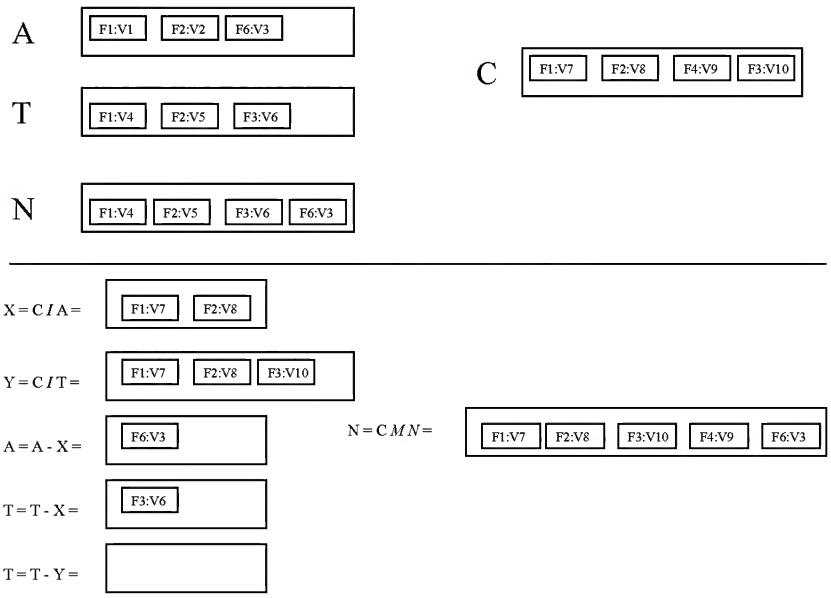
A = A - X

T = T - X

T = T - Y

N = C *M* N

Example.



Assume:

A = Record (UIDi) *E* Locked Change List (X, SP(a))

T = Record (UIDj) *E* UnLocked Change List (X, SP(a))

C = Record (UIDk) *E* Locked Change List (X, SP(b))

The changes made in SP(a) will win over the changes made in SP(b)

N is the original record that holds the most update version of the conflicted record, which most likely resides in a database.

where SP(a), SP(b) *E* SP and X is the database that A, T and C belongs to and UIDi == UIDj == UIDk.

Apply:

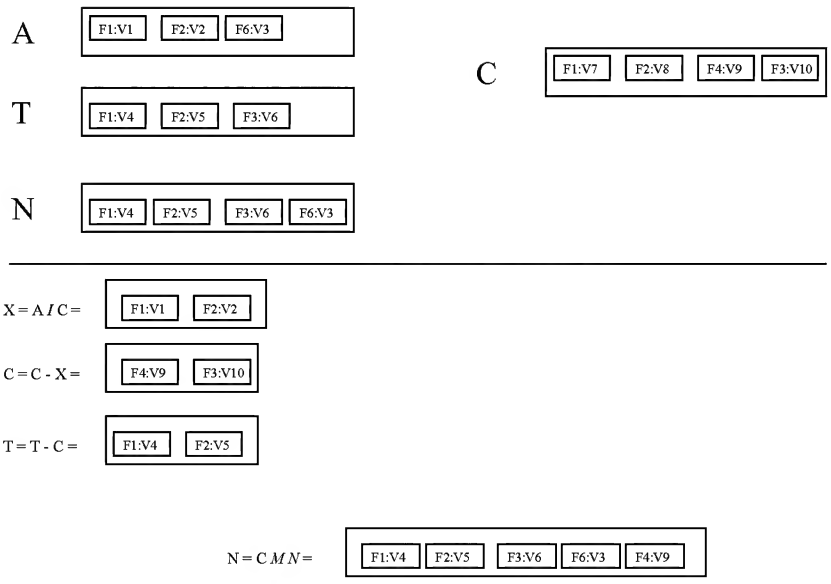
X = A *I* C

C = C - X

T = T - C

N = C *M* N

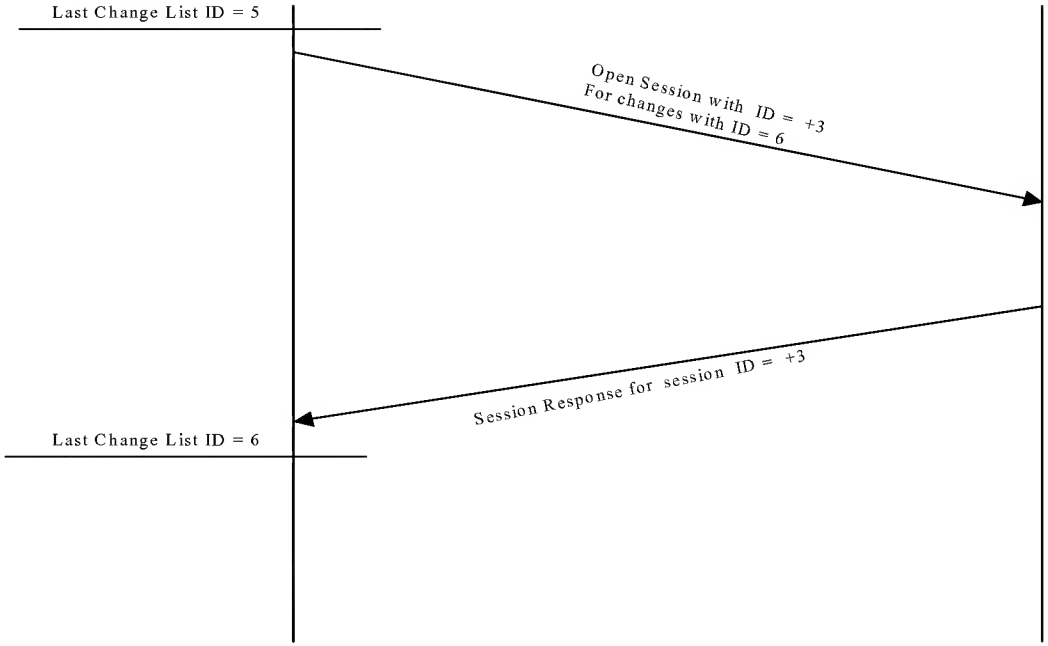
Example.



6.2.4 Updating sync state information

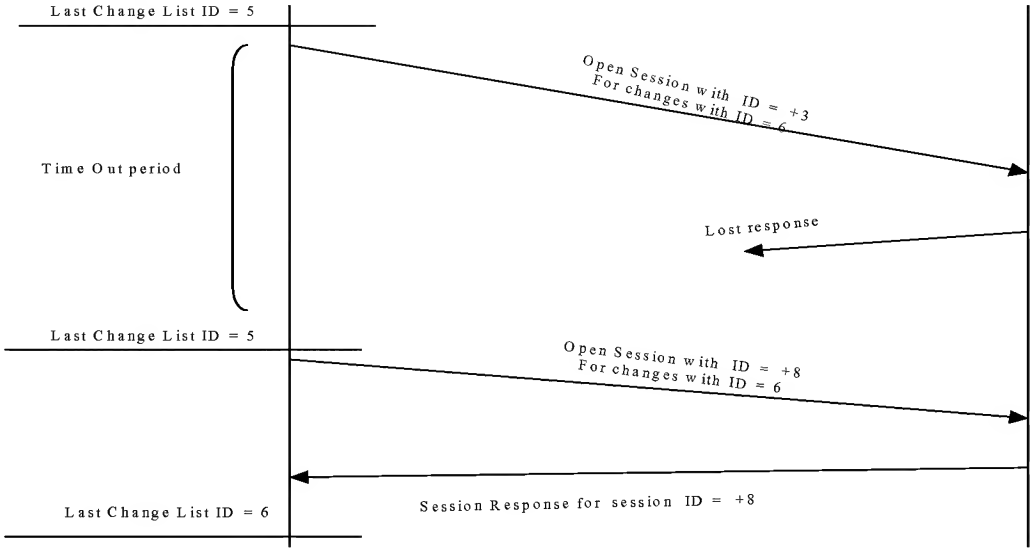
Updating sync state information involves updating the Change List ID and/or the Expected Change List ID. Here is how and when a Sync Party updates its sync state information.

A Sync Party will update its Change List ID whenever it receives a response for a session that was associated with a Change List ID, regardless of the result of the changes. The update process is simply done by incrementing the change List ID. The Sync party might update it is Change List ID before receiving a response for its active session if another session comes from the other sync party which happen to be initiated after the response for the active session has been sent, but it might arrive late response.

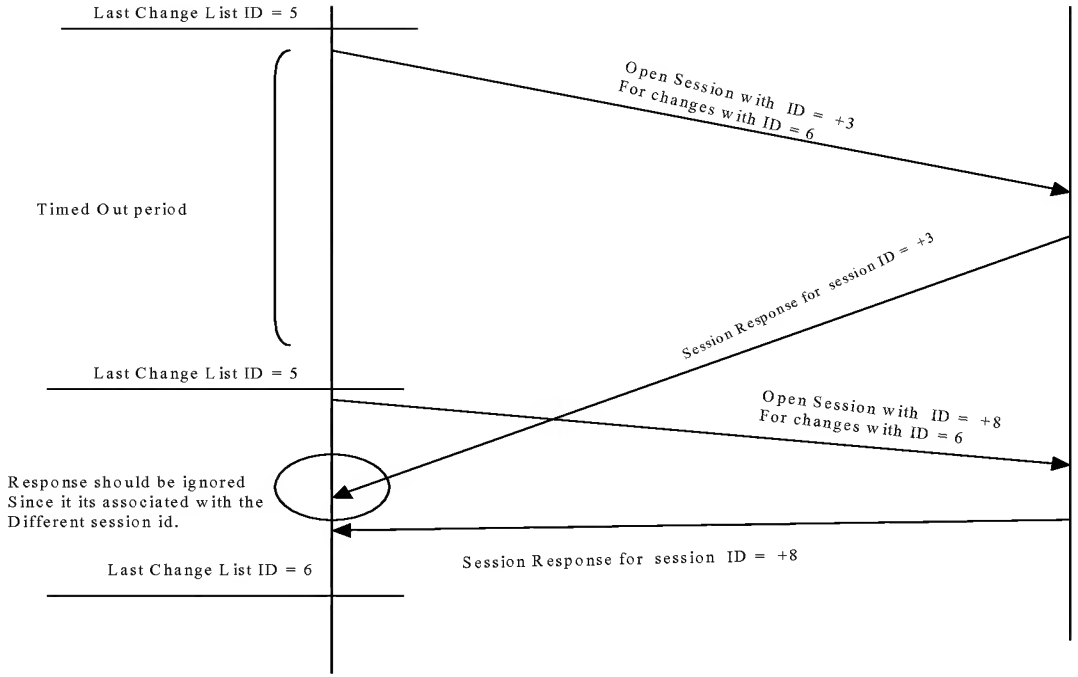

Last Change List ID = 5
Open Session with ID = +3
For changes with ID = 6
Session Response for session ID = +3
Last Change List ID = 6

If a session times out waiting for a response then the same Change List ID is kept for the next session (since the changelist ID has not been received by the recipient). When the list is resent, it must be sent in a new session. So although the Change List ID remains the same, the session ID must be change.

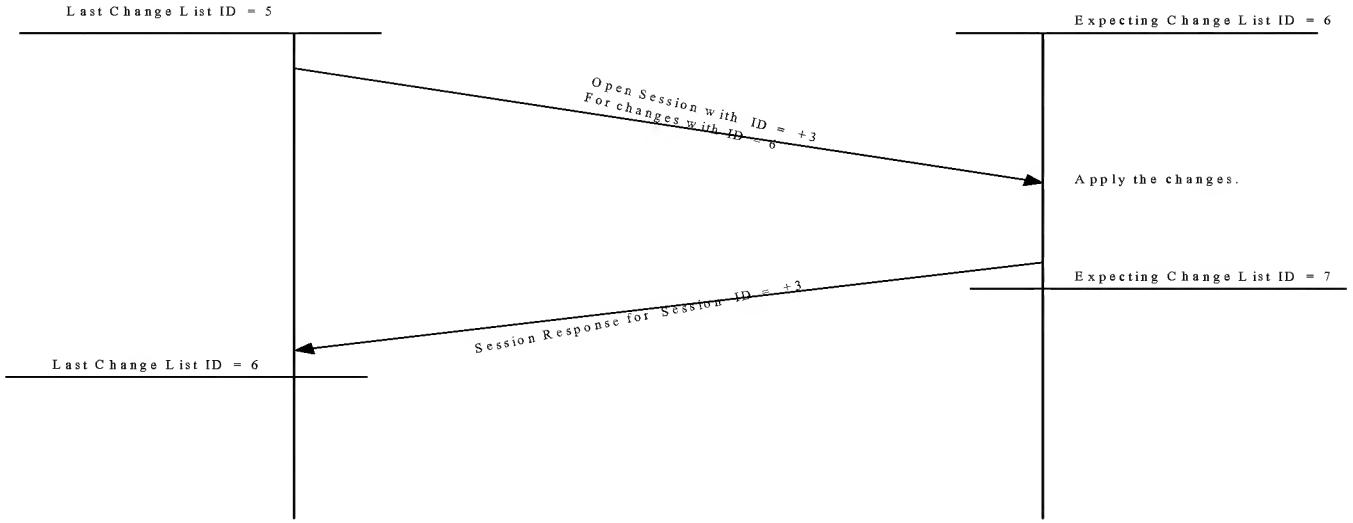
The same Change List ID can be sent in any number of sessions until the recipient acknowledges it. The session ID must ALWAYS be unique and cannot be repeated.



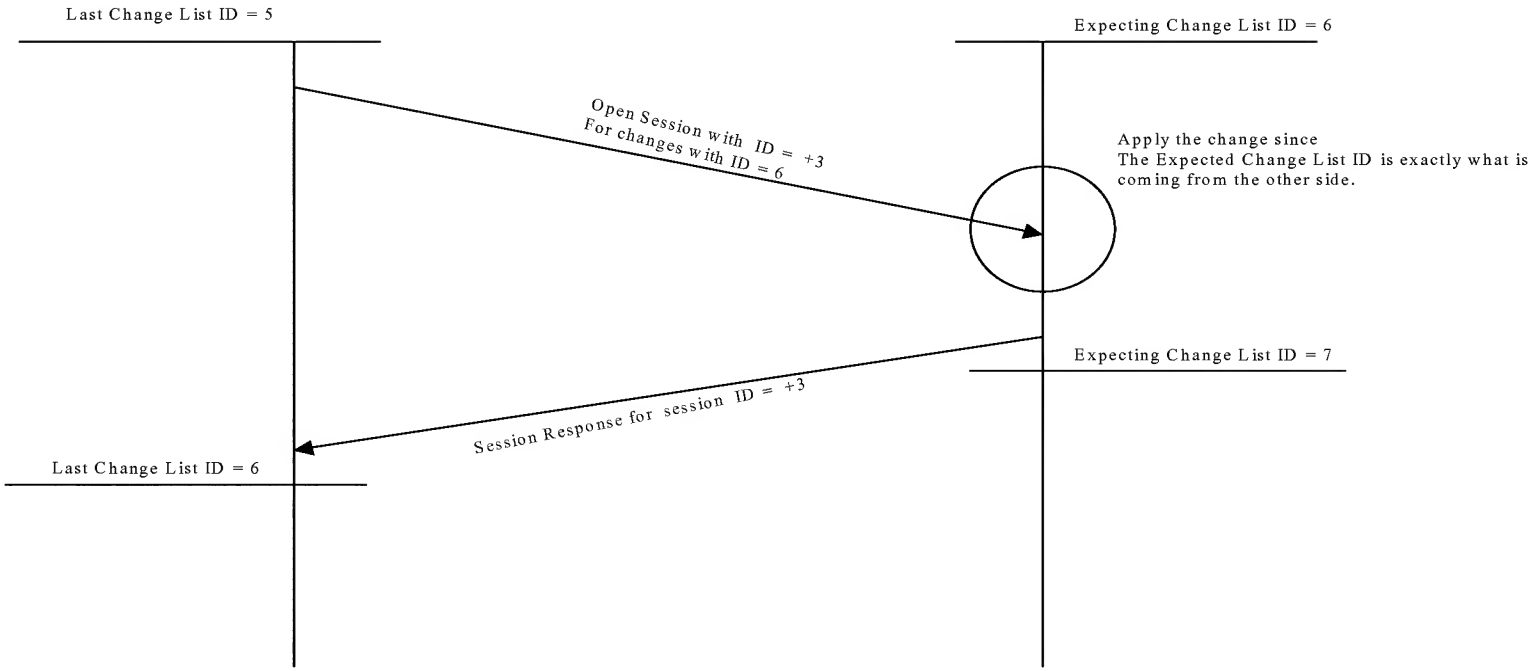
If a response of a session comes late, it should be completely ignored. A Late response is a response that holds a session ID that is no longer active. Generally, a Sync Party should ignore any response to an inactive session or a session that does not exist.



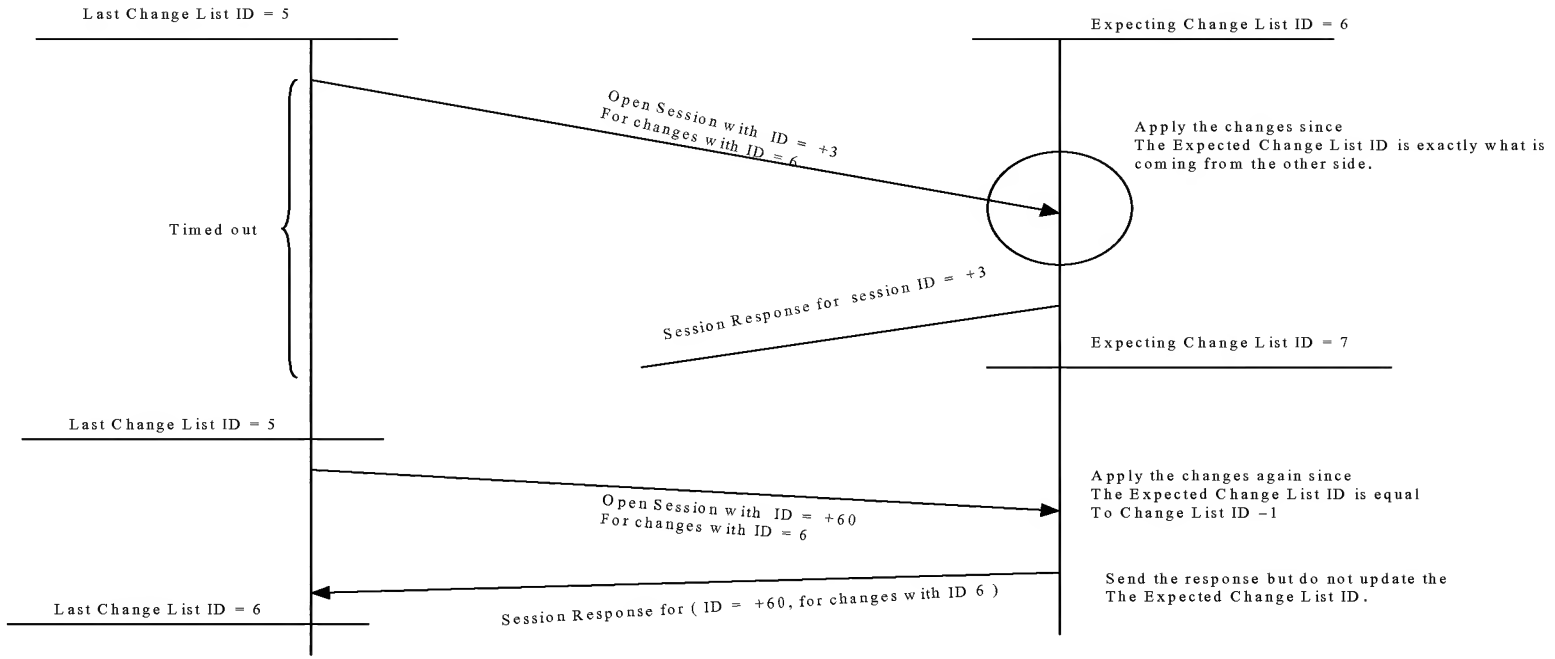
A Sync Party will update its Expected Change List ID whenever it sends a response for a session that was associated with a Change List ID, regardless of the result of the changes that has been sent with the response. The update process is simply incrementing the Expected Change List ID.



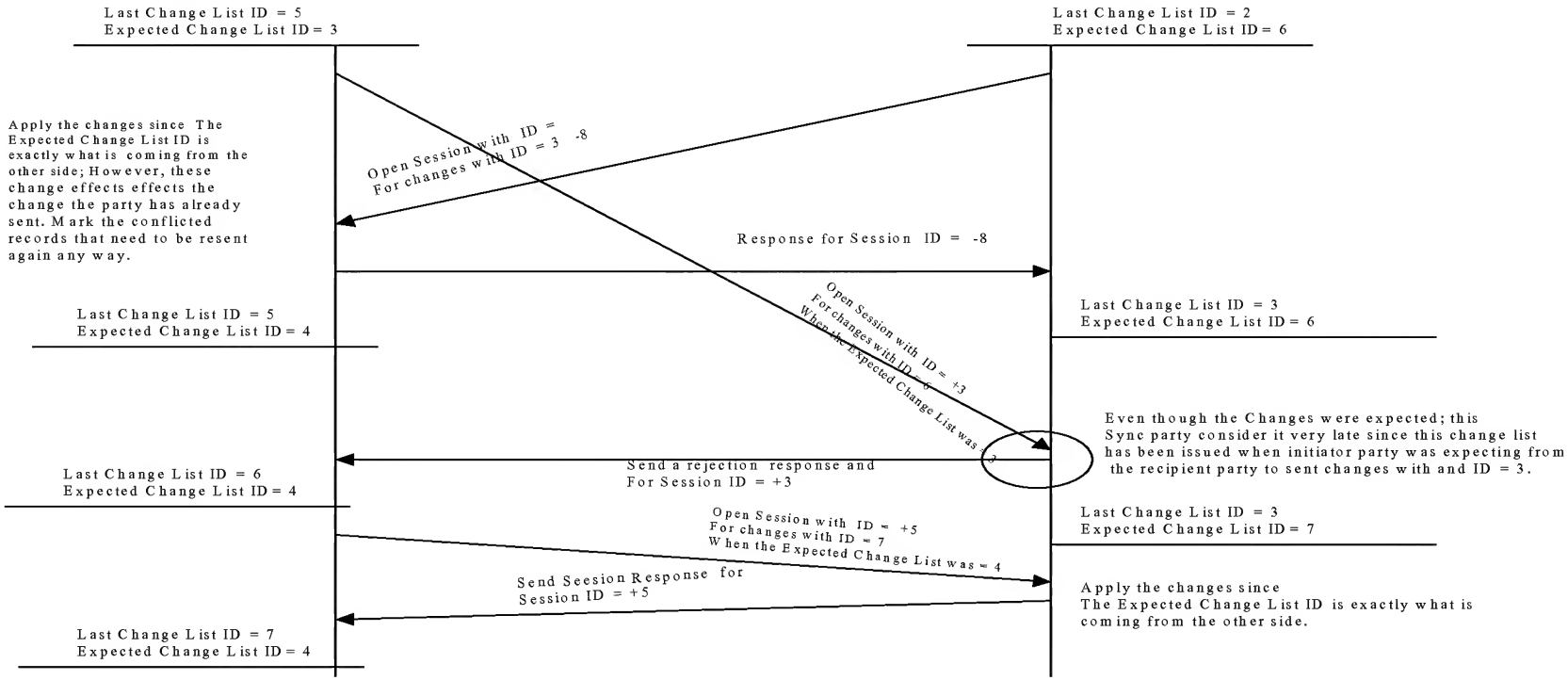
A Sync Party will accept and handle a change list of a session if the sync party was expecting the change list.

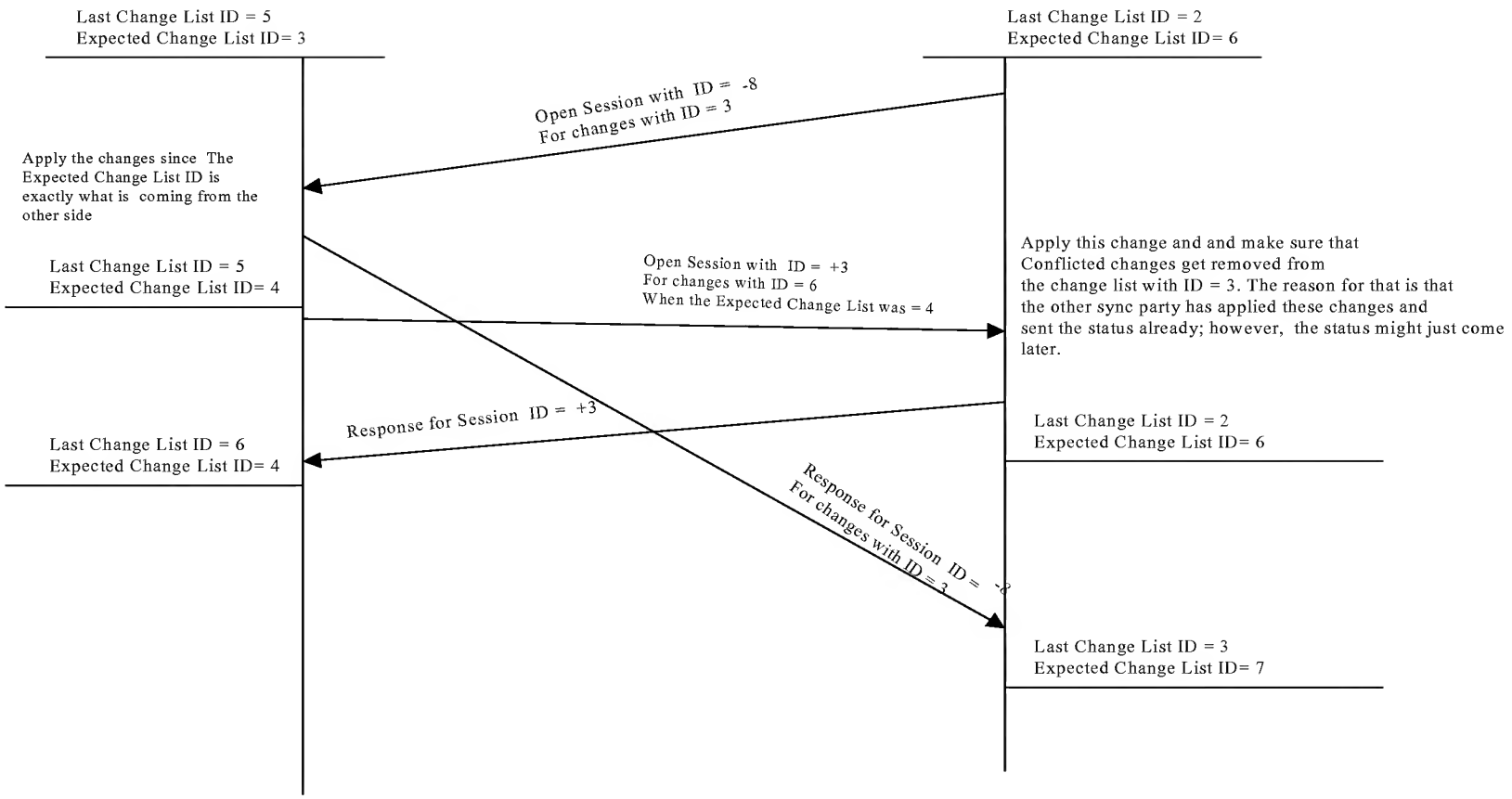


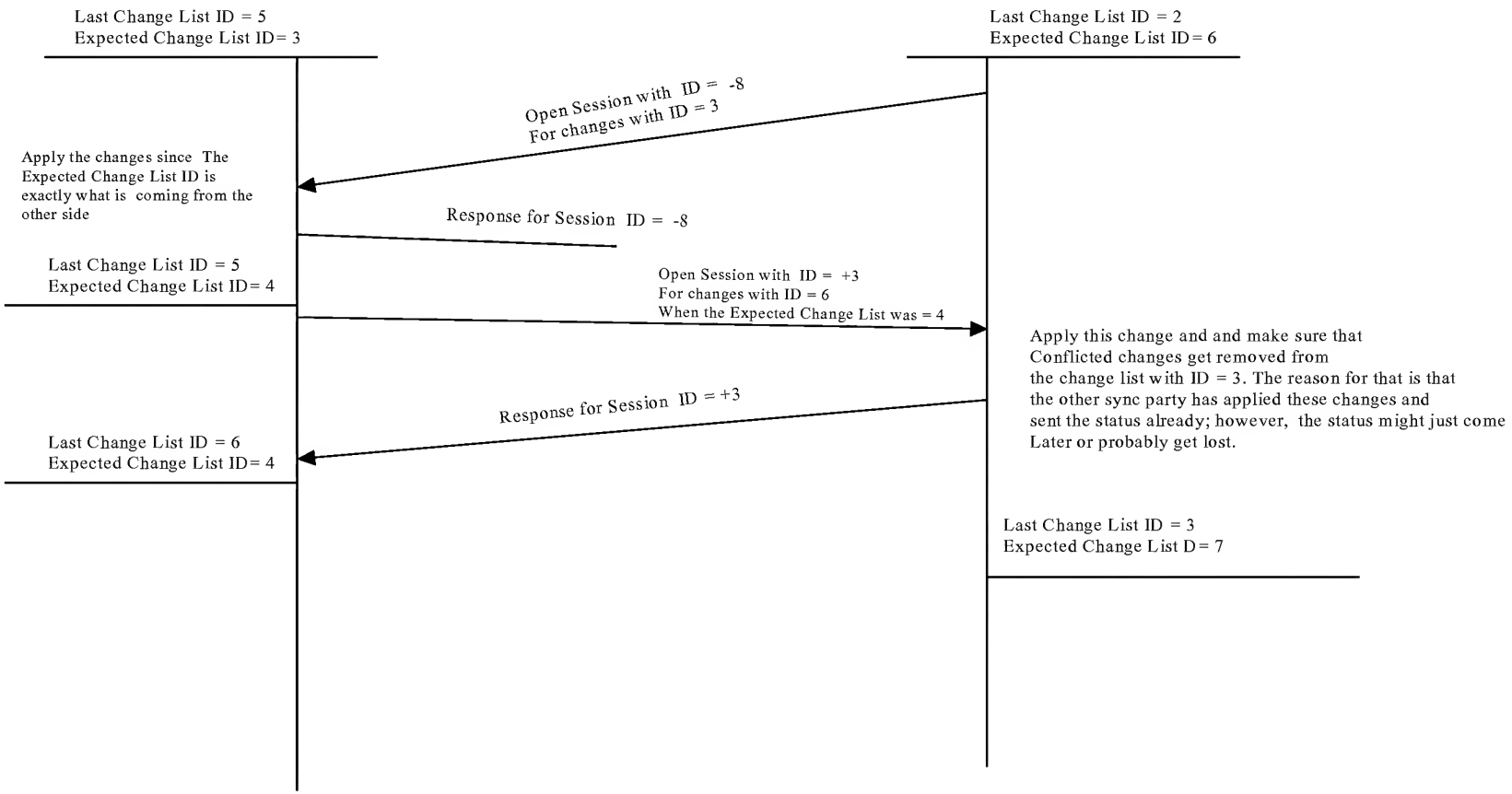
Since Expected Change List ID gets updated anyway regardless of the delivery state of the response, a sync party might decide it should apply the changes again and send the response again if sync party is not able to save the response for a previous session.



A Sync Party will ignore or decline to process a change list if it determines that it carries obsolete changes. This is to avoid conflict between changes. This situation happens after a change list gets sent to the sync party, but it arrives just after the sync party has sent another change list with which it might conflict.




Last Change List ID = 5
Expected Change List ID= 3
Last Change List ID = 2
Expected Change List ID= 6
Open Session with ID = -8
For changes with ID = 3
Apply the changes since The Expected Change List ID is exactly what is coming from the other side
Last Change List ID = 5
Expected Change List ID= 4
Open Session with ID = +3
For changes with ID = 6
When the Expected Change List was = 4
Apply this change and and make sure that Conflicted changes get removed from the change list with ID = 3. The reason for that is that the other sync party has applied these changes and sent the status already; however, the status might just come later.
Response for Session ID = +3
Last Change List ID = 6
Expected Change List ID= 4
Last Change List ID = 2
Expected Change List ID= 6
Response for Session ID = -8
For changes with ID = 3
Last Change List ID = 3
Expected Change List ID= 7


Last Change List ID = 5
Expected Change List ID= 3
Last Change List ID = 2
Expected Change List ID= 6
Open Session with ID = -8
For changes with ID = 3
Apply the changes since The Expected Change List ID is exactly what is coming from the other side
Response for Session ID = -8
Last Change List ID = 5
Expected Change List ID= 4
Open Session with ID = +3
For changes with ID = 6
When the Expected Change List was = 4
Apply this change and and make sure that Conflicted changes get removed from the change list with ID = 3. The reason for that is that the other sync party has applied these changes and sent the status already; however, the status might just come Later or probably get lost.
Response for Session ID = +3
Last Change List ID = 6
Expected Change List ID= 4
Last Change List ID = 3
Expected Change List D= 7

6.2.5 Determining out of sync state

Before a Sync Party applies a change list, it should make sure that its Sync State Information is consistent with the other sync party.

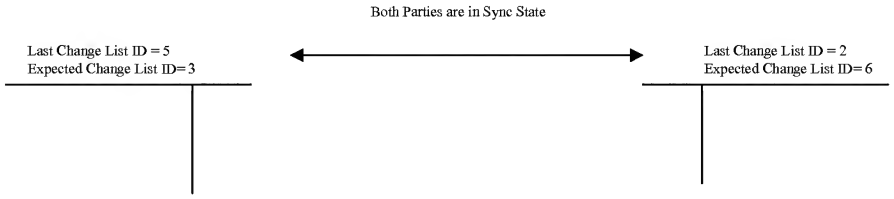
Sync State information of Sync Parties is consistent if Change List Ids are consistent with what sync parties expect from each other.

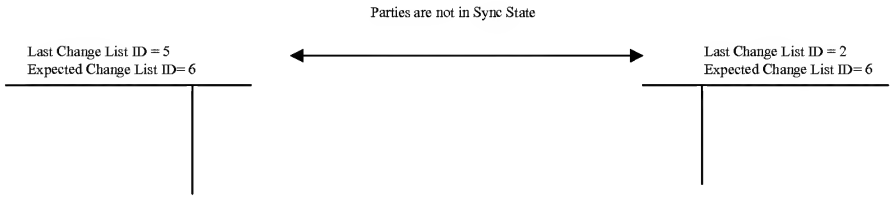
A Sync Party 'A', checks whether it is in Sync State with Sync Party 'B'. If this statement is true:

SyncState(A). Expected Change List ID == SyncState(B). Change List ID

And

SyncState(A).change List id == SyncState(B). Expected Change List ID





If the receiver sync party receives a session, here is how it should deal with the received session.

```
Switch (Receiver. Expected Change List ID – Sender. Current Change List ID)
{
        case –1;
        {
                // Increment the expected change list ID first.  Then Handle the  the Session as in a case 0;
        }

        case +1
        {
            Handle the session and apply the changes as in "case 0" but do not increment the expected change list id.
        }
        break;

        case 0
        {
           // The Receiver is received what was expected

           switch(  Receiver.Last Change List ID - Sender.Expected Change List ID )
           {
                   case –2
                   {
                           Apply the change as in case  -1 and Increment the Receiver Last Change List Id and if the receiver has an active
                           Session, then when a response comes, The Receiver Last Change List ID should not be incremented.
                   }
                   break;

                   case -1
                   {
                                   Just apply the changes from the sender with consideration of the conflicts.
                                   And send the response. Increment the expected change list id.
                   }
                   break;


                   case 0
                   {

                          Sender session MUST be rejected since it holds obsolete changes.
                          Send a response to indicate so.
                   }
                   break;

                   default :
                   {

                       // Verify with the sender before starting slow sync ...
                   }
           }

        } break

        default:
        {
          // Verify with the sender before starting slow sync ...
        }
}
```

The verification process involves sending Verification datagrams between sync parties. When a sync party decides that a received session could not be applied and a slow sync must start, as precautionary step, it should verify that the session is truly sent by the other sync party and the session is not a session that simply happens to be very old one.

A verify datagrams is sent on the same session and if the sender of the session is alive and was expecting a response for the session, then it should replay back with Verification response.

If the Verification Response received then Invalid Sync State get sent back to the session initiator, and the Sync State Information of both sync parties get reset.

If the verification response was not received then the Sync State Information should not be rest.

As soon as a sync party receives invalid sync state as a status of a session, it resets its sync state and simply triggers the slow sync process.

The sender of the invalid sync state status response should reset its sync stat but should not trigger the slow sync process.

How does a Verification datagrams looks like?

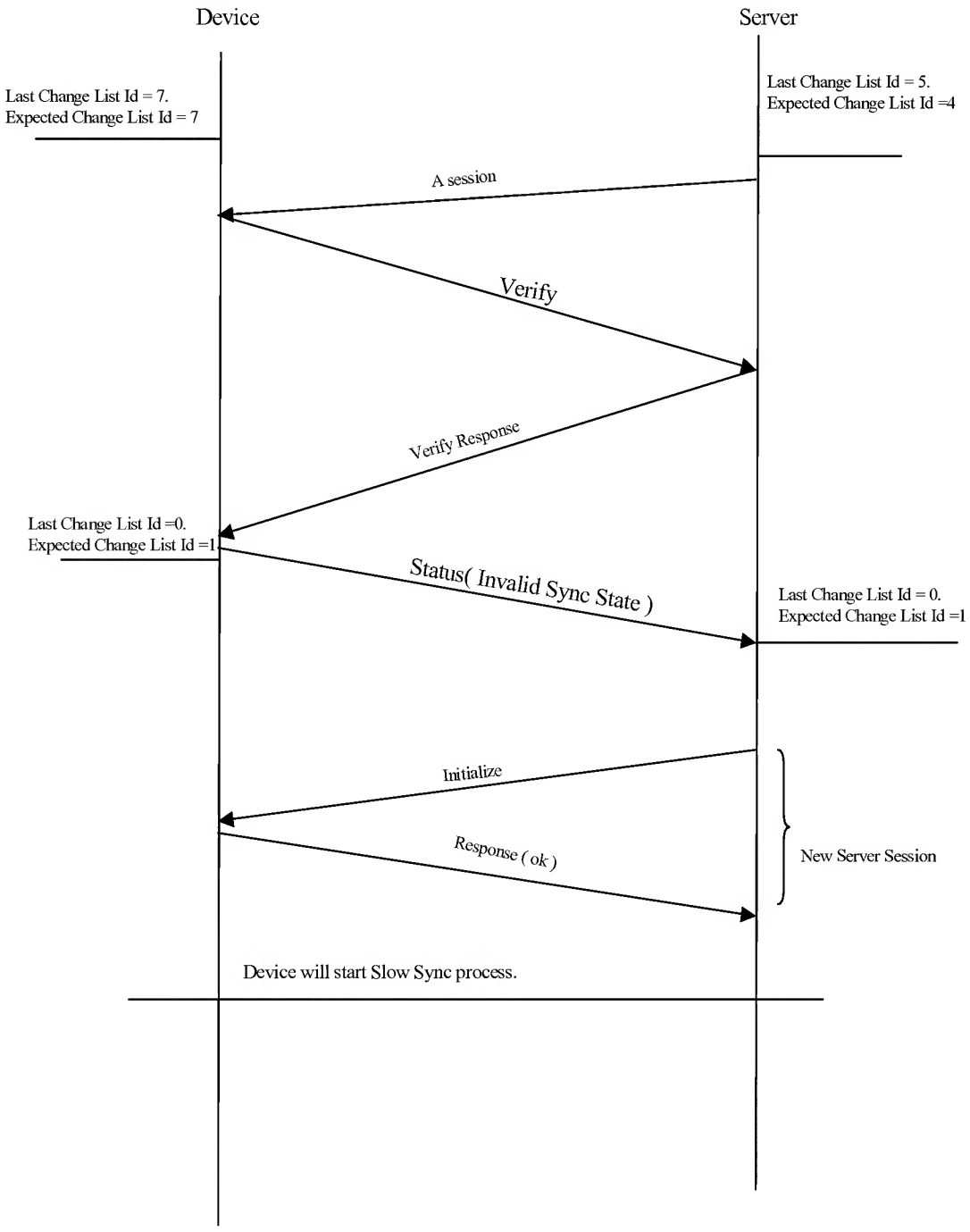
Since verification datagrams a response datagrams, then they only contain Device/Server Session ID and Verification header fields. The Sync Command Section in these Verification Datagrams is always empty one.

A verification request.

```
[0x10]   // Version.
[0x01][0x04]  [0x01][0x02][0x03][0x04] // Device Session being verified.
[0x07][0x01]  [0x00]                   // Verify
[0x00]  /End of the header.
```

A verification response

```
[0x10]   // Version.
[0x01][0x04]  [0x01][0x02][0x03][0x04] // Device Session being verified.
[0x07][0x01]  [0x01]                   // Verify response ( Confirmation ).
[0x00]  /End of the header.
```


Device
Server
Last Change List Id = 7.
Expected Change List Id = 7
Last Change List Id = 5.
Expected Change List Id =4
A session
Verify
Verify Response
Last Change List Id =0.
Expected Change List Id =1
Status(Invalid Sync State)
Last Change List Id = 0.
Expected Change List Id =1
Initialize
Response (ok)
New Server Session
Device will start Slow Sync process.

When a Sync Party decides to "ignore" session because the change list it carries could not be applied, it should send only one response for the whole session to the sender. If the session was sent over multi datagrams, a sync party should send only one response for the whole session, not per every datagram in the session.

A Sync party MUST maintain only one list of ignored sessions.

There are two sessions that should be maintained in ignored sessions list.

- Sessions that are considered out of sync.
- Session that are obsolete.

Obsolete sessions should not be "verified". However, they should be only ignored. Out of sync session should be ignored and verified.

An ignored session should be removed from the list when:

- All the datagrams on the session has been received and discarded.
- A Timeout period elapses since there is a possibility that not all the session datagrams might arrive to the sync party. The timeout period is defined in the configuration section.

6.2.6 Errors and return codes used between sync parties

All the error codes and return values have a permanent effect on a session or a sync command of a session.

For example, if a sync party opens a session with a change list that holds an update for a record that has already been deleted, the return code of that change would be a "Record Not found" and the sync party should not try to send the update again.

Another example would be if deleting a record that has already been deleted where the return code would be "Record Not found" or "SUCCESS".

Not Supported Protocol Version	401	Server/Device
Invalid Session Datagram	402	Server/Device
Invalid Session State	403	Device only
Invalid Sync State	404	Server/Device
Database Not Found	405	Server/Device
Data Source Not Found	406	Server/Device
Not Implemented Command	407	Server/Device
Unknown Command	408	Server/Device
Invalid Command	409	Server/Device
Record Not Found	410	Server/Device
Operation Failure	411	Server/Device
Sync Database Not Enabled	412	Server/Device
Low In Memory	413	Server/Device
Obsolete Session	414	Server/Device
Not Received Datagram	415	Server/Device

6.2.6.1 Not Supported Protocol Version

Error code is returned when the recipient of a session could not handle the protocol version. This error is a global on the whole session. I.e. no more datagram on the session

6.2.6.2 Invalid Session Datagram

Error code is returned when the recipient of a session determines that a datagram is invalid. Datagram is considered invalid if:

- No End if header found
- Current index more than Last Index.
- Any of the indexes holds negative values.
- No Server/Device Session Id exists.

This error is not global on the whole session. The processing of other datagram on the same session might continue, as they might be other valid datagrams on the session.

6.2.6.3 Invalid Session State

Error code is returned during the processing sync datagram commands where the order of the commands and the dependency relation ship between commands might be broken. This Error is based per command as the reset of the commands might have the correct order and dependency relationship between them.

6.2.6.4 Invalid Sync State

Error code is returned if a sync party detects that a slow sync process is required. Both the recipient and sender of this error should reset their sync state information to

- Change List Id to 0
- Expected Change List Id to 1;

This error should not be sent unless the verification step is done first.

6.2.6.5 Database Not Found

This error is returned per **USE** command bases. If a USE command in a datagram points to a database that does not exist in the system, all sync commands that depend on it should not be executed

6.2.6.6 Data Source Not Found

This error is returned per **USE** command bases. If a USE command in a datagram points to a database that does not exist in the system, all sync commands that depend on it should not be executed

6.2.6.7 Not Implemented Command

This error is returned per command in a sync datagram. If a command is not implemented, then this error is returned and the process of executing other commands in a datagram that do not depend on it should continue.

6.2.6.8 Unknown Command

This error is returned per command in a sync datagram. If a command is not defined, then this error is returned and the process of executing other commands in a datagram should continue.

6.2.6.9 Invalid Command

This error is returned per command in a sync datagram. If a command is not valid, then this error is returned and the process of executing other commands in a datagram should continue. The validation of a sync command depends on the command and what type of mandatory parameters and their valid values in a command. No need to send more details as to what made a command is invalid; however logging the details would be helpful.

6.2.6.10 Record Not Found

This error is returned per command. If a valid Update command being executed on a record that no longer exists, then this error is returned.

6.2.6.11 Operation Failure

This error is returned per command. If a general failure occurs during the execution of a sync command, then this error is returned. All sync commands that depend on the failed command should not be executed.

6.2.6.12 Sync Database Not Initialized

This error is returned per sync command. If database not initialized yet and a sync party tries to do a sync operation on it, except during the initialization phase, this error should be returned. The expected reaction to this error is to initialize database again.

6.2.6.13 Low In Memory

This error is returned per sync command. If no enough memory or memory is low, then just this error is returned. No reaction to this error is defined.

6.2.6.14 Obsolete Session

This error is returned per session. This error indicates that the session carries old changes and the changes cannot be applied at this time.

6.2.6.15 Not Received Datagram

This error will be sent if a session times out waiting for a datagram/datagrams to be received. The changes that carried by not received datagrams must be sent again on different session.

7 Data Sync Examples

(More will be added to this).

Request

```
Header Session
{
      Version = 0x10
      Session.ID = 104567845
      ChangeList.ID = 6
      ExpectedChangeList.ID = 5
}

Sync Commands Session
{
      Use    Exchange,  Address Book
      Add   Record.UID=102345   Record.FIELDS = {  Fname1, Lname1, Rim }
      Add   Record.UID=102346   Record.FIELDS = {  Fname2, Lname2, Rim }
      Add   Record.UID=102347   Record.FIELDS = {  Fname3, Lname3, Rim }
      Add   Record.UID=102365   Record.FIELDS = {  Fname4, Lname4, Rim }
}
```

Response

```
Header Session
{
      Version = 0x10
      Session.ID = 104567845
}
Sync Commands Session
{
     Empty  // if every thing went well in the session.
}
```

```
Header Session
{
      Version = 0x10
      Session.ID = 1045678845
}

Sync Commands Session
{
      Status ErrorCode = 419 // Obsolete Session
}
```

```
Header Session
{
      Version = 0x10
      Session.ID = 1045678845
}

Sync Commands Session
{
      // Note '2' and '3' consists of 'Datagram Seq#.Index of Command'
      // See 'Cmd.ID' in 'Sync Command's Parameters'
      Status [(2, 412), (3, 413)]
}
```

8 Appendix

8.1 Service Book Information

The OTA PIM synchronization service book consists of the Content ID "SYNC" and the following configuration data shown below. All fields are type-length encoded. Note that any field starting with "*", indicates it must be present at least once.

Service Book Configuration Field	Type Byte code	Values	Optional
Default Service	0x01	Boolean to indicate if this service is the default service.	Yes

8.2 Configuration Settings Information

The following configuration information is sent from the Sync Server to the device in response to the 'Get Configuration' command. All fields are type-length encoded. Note that any field starting with "*", indicates it must be present at least once.

Configuration Settings			
Field	Type Byte code	Values	Optional
Protocol Version	0x01	Sync Protocol Version supported by the Sync Server.	No
*Data Source	0x02	Data Source object.	No
Session Time-out	0x03	Session time in minutes. Defaults to 5 Min.	Yes
Batch Sync Time	0x04	Batch sync time in minutes. Defaults to 5 Min.	Yes
OTA User State	0x05	Indicates if user enabled or disabled for OTA PIM sync.	No
Number Of Retries	0x06	Indicates the number of retries to send a packet. Defaults to 3.	Yes
Ignored Session Timeout	0x07	Timeout for the ignored sessions. Defaults to 30 Min.	Yes

Data Source			
Field	Type Byte code	Values	Optional
Version	0x01	Data Source version	No
Data Source Name	0x02	Data source name. For example, "Notes", "Exchange" or "Backup"	No
Data Source ID	0x03	An ID used for optimization purposes.	No
Data Base	0x04	Data base object	No
Default	0x05	Default data source. Defaults to false;	Yes

Data Base Field	Type Byte code	Values	Optional	Defaults
Version	0x01	The version of the database	No	
Database Name	0x02	The database name on the device to sync with. The database name is associated with a Data Source.	No	
Database ID	0x03	An Id that would be used instead of the name	No	
Sync Type	0x04	0x00 - Disabled 0x01 - 1 Way Sync To Device 0x02 - 1 Way Sync to Server 0x03 - 2 Way Sync to server.	Yes	0x03
Conflict Resolution	0x05	0x00 = Server Wins; 0x01 = Device Wins.	Yes	0x00
Schema	0x06	Contains a series of mapped record tags containing Type and Unique # Tag(s).	No	See below for this field format.
Sync Flags	0x07	A series of bit flags: AllowOTASlowSync:1 EraseOnSlowSync:2	Yes	0x01
Sync Mode	0x08	The Sync mode for this database. The possible values are 0x00 for BATCH SYNC MODE and 0x01 for PROGRESSIVE Sync MODE.	Yes	0x00
Number Of Groups	0x09	The number of groups applied during the slow sync process	No	0xFF

Schema field payload format

A tag represents every column in the database. All tags must be unique and thus all columns are unique within a database. Every column representation in the schema field is represented in TLE format:

[Column Tag] [[Length]] [[Type] Optional ([Unique Tags])], where

- Column Tag is the unique tag for the column
- Type is the column data type, which could be one of the following values:
  - BINARY = 0x00
  - CString = 0x01
  - UnicodeCString = 0x02

The type field indicates as well whether or not the field is a Key field in the record.

Thus bits from 0 to 6 indicates the type and the bit number 7,MSB, identifies whether Field is a key or not.

Optional Unique Tags bytes to fix the RIM Legacy applications where the schema of a database could have columns with tags that are not unique. So to correct the schema, unique tags is used to re-map the columns to make the schema's columns -unique.

```
Example 1
[ 0x15 ] [ 0x01 ]  [ [ 0x00 ] ] ;
[ 0x13 ] [ 0x01 ]  [ [ 0x01 ] ] ;
[ 0x20 ] [ 0x01 ]  [ [ 0x01 ] ] ;

Note that the database schema hold unique tags, and total number of
Columns is 3

Example 2
[ 0x15 ] [ 0x04 ]  [ [ 0x00 ] [0x016][0x17][0x18]] ;
[ 0x13 ] [ 0x01 ]  [ [ 0x01 ] ] ;
[ 0x20 ] [ 0x01 ]  [ [ 0x01 ] ] ;

Note that the database schema does not hold unique tags, and total
number of Columns is 6. And the schema correction makes the schema
look like

0x15, 0x16, 0x17, 0x18, 0x13, 0x20

rather than

0x15, 0x15, 0x15, 0x15, 0x13, 0x20
```